



# **Guide to BASIC**

**Version 3.11**

**Model "D"**

**Leading Edge Hardware Products, Inc.**





**Guide to BASIC  
Version 3.11**

**for the  
Model " D "**

**Leading Edge Hardware Products, Inc.  
Canton, MA**



**First Edition: February, 1986**  
**Version 3.11**

Information in this document is subject to change without notice and does not represent a commitment on the part of Leading Edge Hardware Products, Inc. or Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft GWBASIC® or the MS-DOS® Operating System on disk, or on any other medium for any purpose other than the purchaser's personal use.

**Microsoft GWBASIC version 3.11**  
**Copyright 1983, 1984, 1985 Microsoft Corporation**

From time to time, changes are made to the information contained in this user's guide. Any such changes are incorporated in new editions of this user's guide. You should check with your dealer from time to time to see if new editions are available and the information therein is of interest.

As modified by Leading Edge Hardware Products, Inc. and Phoenix Software Associates Ltd.

**Copyright 1986, Leading Edge Hardware Products, Inc.**  
**Copyright 1984, 1985 Phoenix Software Associates Ltd.**

Leading Edge and logo are registered trademarks of Leading Edge Products, Inc.  
GWBASIC and MS-DOS are registered trademarks of Microsoft Corporation.

Printed in the United States of America

## MODEL D MANUALS: WHERE TO TURN FIRST

### OPERATOR'S GUIDE

Read this guide first, as it serves as an introduction to the Model D. It explains how to set up and use the computer, and how to solve any problems you may encounter. Turn next to the *MS-DOS User's Guide*.

### MS-DOS USER'S GUIDE

You need an operating system to run the Model D. This guide contains the basics that you need to know to run the Model D with the MS-DOS operating system. Procedures are presented in an easy-to-follow, step-by-step format. Read through the entire guide (it's short) before you begin to use MS-DOS.

### MS-DOS REFERENCE MANUAL

This manual describes the features of the MS-DOS operating system in a more technical format. You'll discover more powerful ways to use MS-DOS. Read the *Reference Manual* after you're comfortable using MS-DOS and are familiar with the basic commands outlined in the *MS-DOS User's Guide*.

### GUIDE TO BASIC

Before you can write BASIC programs on the Model D, you'll need to learn about GWBASIC. This book explains GWBASIC commands, statements, functions, and variables and the correct syntax for each. **It does not teach you how to program in GWBASIC.**





## TABLE OF CONTENTS

### Preface

How To Use This Guide.....	1
Major Features of Leading Edge GWBASIC.....	2
Syntax Notation.....	3

### Chapter 1: General Information On GWBASIC

Programming.....	1-1
Starting BASIC.....	1-1
Definitions of Start Options.....	1-2
Redirection of Standard Input and Standard Output.....	1-4
Entering GWBASIC Commands.....	1-5
How to Enter a Program.....	1-6
How to Make Corrections to the Current Program.....	1-7
The Screen Editor.....	1-9
Screen Editor Corrections.....	1-9
Screen Editing Keys and Their Functions.....	1-9
Modes of Operation.....	1-10
Line Format.....	1-11
Line Numbers.....	1-11
Using Characters.....	1-11
The Character Set.....	1-11
Reserved Words.....	1-12
Constants.....	1-14
Single and Double Precision Form for Numeric Constants.....	1-15
Variables.....	1-15
Variable Names and Declaration Characters.....	1-16
Array Variables.....	1-16
Space Requirements.....	1-17
Type Conversion.....	1-17
Expressions and Operators.....	1-18
Arithmetic Operators.....	1-19
Integer Division and Modulo Arithmetic.....	1-20
Overflow and Division by Zero.....	1-20
Relational Operators.....	1-21
Logical Operators.....	1-21
Functional Operators.....	1-24
String Operations.....	1-24
Error Messages.....	1-24
Files.....	1-25
Device Names.....	1-26



MS-DOS File System.....	1-26
Directory Paths.....	1-27
Using The Screen.....	1-27
Display Types.....	1-28
Graphics Mode.....	1-29
Specification of Coordinates.....	1-30
The Keyboard.....	1-31
Typewriter Keyboard.....	1-31
Function Keys.....	1-33
Numeric Keypad.....	1-34
Special Key Combinations.....	1-34
GWBasic Program Editor.....	1-34
Special Program Editor Keys.....	1-35

## Chapter 2: GWBasic Commands and Statements

AUTO.....	2-2
BEEP.....	2-3
BLOAD.....	2-4
BSAVE.....	2-5
CALL.....	2-6
CHAIN.....	2-7
CHDIR.....	2-9
CIRCLE.....	2-10
CLEAR.....	2-11
CLOSE.....	2-12
CLS.....	2-13
COLOR (Text).....	2-14
COLOR (Graphics).....	2-16
COM.....	2-17
COMMON.....	2-18
CONT.....	2-19
DATA.....	2-20
DEF FN.....	2-21
DEF INT/SNG/DBL/STR.....	2-22
DEF SEG.....	2-23
DEF USR.....	2-24
DELETE.....	2-25
DIM.....	2-26
DRAW.....	2-27
EDIT.....	2-30
END.....	2-31
ENVIRON.....	2-32
ERASE.....	2-34
ERROR.....	2-35
FIELD.....	2-37
FILES.....	2-38
FOR...NEXT.....	2-39
GET.....	2-41
GET (Graphics).....	2-42
GOSUB...RETURN.....	2-44
GOTO.....	2-45
IF.....	2-46
INPUT.....	2-48
INPUT#.....	2-50
IOCTL.....	2-51
KEY.....	2-52
KEY [n] ON/OFF/STOP.....	2-54
KILL.....	2-56

LCOPY.....	2-57
LET.....	2-58
LINE.....	2-59
LINE INPUT.....	2-61
LINE INPUT#.....	2-62
LIST.....	2-63
LLIST.....	2-65
LOAD.....	2-66
LOCATE.....	2-67
LPRINT and LPRINT USING.....	2-68
LSET and RSET.....	2-69
MERGE.....	2-70
MID\$.....	2-71
MKDIR.....	2-72
NAME.....	2-73
NEW.....	2-74
ON COM.....	2-75
ON ERROR GOTO.....	2-76
ON...GOSUB and ON...GOTO.....	2-77
ON KEY.....	2-78
ON PEN.....	2-81
ON PLAY.....	2-82
ON STRIG.....	2-83
ON TIMER.....	2-85
OPEN.....	2-86
OPEN COM.....	2-88
OPTION BASE.....	2-91
OUT.....	2-92
PAINT.....	2-93
PALETTE.....	2-96
PALETTE USING.....	2-97
PEN.....	2-98
PLAY.....	2-100
POKE.....	2-103
PRINT.....	2-104
PRINT USING.....	2-106
PRINT# and PRINT# USING.....	2-110
PSET and PRESET.....	2-112
PUT (Files).....	2-113
PUT (COM).....	2-114
PUT (Graphics).....	2-115
RANDOMIZE.....	2-117
READ.....	2-118
REM.....	2-120
RENUM.....	2-121
RESET.....	2-122
RESTORE.....	2-123
RESUME.....	2-124
RETURN.....	2-125
RMDIR.....	2-126
RUN.....	2-127
SAVE.....	2-128
SCREEN.....	2-129
SHELL.....	2-131
SOUND.....	2-134
STOP.....	2-135
STRIG.....	2-136
SWAP.....	2-137
SYSTEM.....	2-138



TIMES\$.....	2-139
TIMER.....	2-140
TRON and TROFF.....	2-141
VIEW.....	2-142
WAIT.....	2-143
WHILE...WEND.....	2-144
WIDTH.....	2-145
WINDOW.....	2-147
WRITE.....	2-149
WRITE#.....	2-150

### Chapter 3: GWBASIC Functions and Variables

ABS.....	3-2
ASC.....	3-2
ATN.....	3-3
CDBL.....	3-3
CHR\$.....	3-4
CINT.....	3-5
COS.....	3-5
CSNG.....	3-6
CSRLIN.....	3-6
CVI, CVS, CVD.....	3-7
DATE\$.....	3-8
ENVIRON\$.....	3-8
EOF.....	3-9
ERDEV and ERDEV\$.....	3-9
ERR and ERL.....	3-10
EXP.....	3-11
FIX.....	3-11
FRE.....	3-12
HEX\$.....	3-12
INKEY\$.....	3-13
INP.....	3-13
INPUT\$.....	3-14
INSTR.....	3-14
INT.....	3-15
IOCTL\$.....	3-16
LEFT\$.....	3-17
LEN.....	3-17
LOC.....	3-18
LOF.....	3-19
LOG.....	3-20
LPOS.....	3-20
MID\$.....	3-21
MKI\$, MKS\$, MKD\$.....	3-22
OCT\$.....	3-22
PEEK.....	3-23
PLAY(n).....	3-24
PMAP.....	3-24
POINT.....	3-25
POINT (function).....	3-25
POS.....	3-26
RIGHT\$.....	3-26
RND.....	3-27
SCREEN.....	3-27
SGN.....	3-28
SIN.....	3-29
SPACE\$.....	3-29

SPC.....	3-30
SQR.....	3-30
STICK.....	3-31
STR\$.....	3-32
STRIG.....	3-33
STRINGS\$.....	3-34
TAB.....	3-34
TAN.....	3-35
TIMES\$.....	3-36
TIMER.....	3-37
USR.....	3-37
VAL.....	3-38
VARPTR.....	3-39
VARPTR\$.....	3-41

## Appendix A: GWBASIC Disk I/O

Program File Commands.....	A-1
Protected Files.....	A-2
Disk Data Files — Sequential and Random I/O.....	A-2
Sequential Files.....	A-2
Program 1 — Creating a Sequential Data File.....	A-4
Program 2 — Accessing a Sequential File.....	A-5
Adding Data to a Sequential File.....	A-5
Random Files.....	A-6
Creating a Random File.....	A-6
Program 3 — Creating a Random File.....	A-7
Accessing a Random File.....	A-7
Program 4 — Accessing a Random File.....	A-8
Program 5 — Inventory.....	A-8

## Appendix B: Communications

Communications.....	B-1
---------------------	-----

## Appendix C: GWBASIC Assembly Language Subroutines

Memory Allocation.....	C-1
The CALL Statement.....	C-2
User Function Calls.....	C-5

## Appendix D: Example of Execution

Start GWBASIC.....	D-1
--------------------	-----

## Appendix E: Converting Programs To Leading Edge GWBASIC

String Dimension.....	E-1
Multiple Assignments.....	E-2
Multiple Statements.....	E-2
MAT Functions.....	E-2

## Appendix F: Summary of Error Codes and Error Messages

Messages.....	F-1
---------------	-----

## Appendix G: Mathematical Functions

Derived Functions.....	G-1
------------------------	-----

## Appendix H: ASCII Character Codes

Special Codes in GWBASIC.....	H-1
ASCII Codes.....	H-2
Extended Codes.....	H-4

## Appendix I: Syntax List

Commands.....	I-2
File.....	I-2
Program Execution.....	I-2
Program Creation.....	I-2
Output to Printer.....	I-3
Return to MS-DOS.....	I-3
Statements.....	I-3
Non I/O.....	I-3
Screen.....	I-5
Music.....	I-6
Input/Output (Terminal).....	I-6
Port.....	I-7
Input/Output (File).....	I-7
Function Key.....	I-8
Communications.....	I-8
Error Handling.....	I-8
Debugging.....	I-8
Pen Handling.....	I-9
Functions and Variables.....	I-9
Arithmetic.....	I-9
String Handling.....	I-10
Input.....	I-11
File Status.....	I-11
Miscellaneous.....	I-11

## Appendix J: Key Scan Codes

Hexadecimal Codes.....	J-1
------------------------	-----

## Index

## **PREFACE**

Leading Edge's GWBASIC interpreter is an advanced version of Microsoft Corporation's BASIC interpreter. Throughout this manual the term GWBASIC refers to this product.

## **HOW TO USE THIS GUIDE**

In order to use this guide, you should have some general knowledge of computer programming concepts. This guide does not attempt to teach you *how* to program a computer in GWBASIC.

This guide contains three Chapters plus nine Appendices:

### **Chapter 1    General Information on GWBASIC**

This chapter tells you what you need to know to begin using GWBASIC. Included in this chapter is information about the keyboard characters, filenames, and how data and relationships are expressed using GWBASIC.

### **Chapter 2    GWBASIC Commands and Statements**

This chapter is a reference section containing each of the GWBASIC commands and statements in alphabetical order. It contains the syntax and examples of correct usage for each.

### **Chapter 3    GWBASIC Functions and Variables**

This chapter is a reference section containing each of the GWBASIC functions and variables in alphabetical order. It contains the syntax and examples of correct usage for each.

**Appendices**    Appendix A through Appendix J contain additional information you may find useful, such as disk input and output, communications, GWBASIC assembly language support, error codes, mathematical functions, a syntax list, and the ASCII character set. Also included are sections of interest to an experienced programmer.

We suggest that you read Chapter 1 very carefully to become familiar with GWBASIC. Chapters 2 and 3 can be used to find information you need when you are actually programming.

## MAJOR FEATURES OF LEADING EDGE GWBASIC

The following features are included in this version of GWBASIC:

- o Four variable types: Integer (+32767), String (up to 255 characters), Single Precision Floating Point (7 digits), Double Precision Floating Point (16 digits)
- o Double Precision Transcendentals. Optional with the /D switch
- o Trace facility (TRON/TROFF) for easy debugging
- o Error trapping using the ON ERROR GOTO statement
- o Event trapping is enabled
- o PEEK and POKE statements to read and write to any memory location
- o Automatic line numbering and renumbering, including referenced line numbers
- o Boolean operators OR, AND, NOT, XOR, EQV, IMP
- o Formatted output using the complete PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, comma insertion
- o Direct access to 64KB I/O ports with the INP and OUT functions
- o Extensive program editing facilities via screen editor
- o User definable keyboard trapping
- o Assembly language subroutine calls are supported
- o IF/THEN/ELSE and nested IF/THEN/ELSE constructs
- o GWBASIC supports variable length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE
- o Extended character sets
- o Sound generation and a feature that plays sequences of programmed notes
- o An internal clock that keeps track of the time and date
- o Asynchronous communication (RS-232C) is supported
- o Redirection of Standard Input (INPUT, LINE INPUT) and Standard Output (PRINT)
- o Tree structured directories for better disk organization

- o Improved Disk I/O facilities for handling larger files
- o Directory management support (MKDIR, CHDIR, RMDIR)
- o Improved Graphics: Line Clipping, VIEW, WINDOW
- o More precise control of GWBASIC's memory allocation for your routines with the /M: switch
- o Hercules compatible monochrome graphics
- o 16 color high resolution graphics

## SYNTAX NOTATION

The following notation is used throughout this guide in descriptions of command and statement syntax and in examples of GWBASIC code:

[ ] Square brackets indicate that the enclosed entry is optional

*italics* Words in italics indicate variable data that the user must enter, for example, *filename*. Sometimes the entries shown are optional.

... Ellipses indicate that an entry may be repeated as many times as needed or desired.

CAPS Capital letters indicate portions of statements or commands that must be entered exactly as shown.

All other punctuation (such as commas, colons, slash marks, parentheses, and equal signs) must be entered exactly as shown.





## CHAPTER 1

### GENERAL INFORMATION ON GWBASIC

#### PROGRAMMING

##### Starting GWBASIC

To use GWBASIC to create or edit a program, you must follow these steps:

1. Insert the MS-DOS system disk into Drive A and turn the computer ON.
2. When the MS-DOS prompt (A>) is displayed, remove the MS-DOS disk and insert the BASIC disk in Drive A.
3. To start programming, type:

**basic**

and then press **RETURN**.

You can modify GWBASIC's programming environment by specifying start options after you type **BASIC** on the command line. These start options are:

```
BASIC [filename ]  
BASIC [ < stdin ][>][> stdout ]]  
BASIC [/C:combuffer size ]  
BASIC [/D]  
BASIC [/F:files ]  
BASIC [/M:highest memory location ][, maximum block size ]  
BASIC [/S:buffer size ]
```

To exit GWBASIC and return to MS-DOS, type

**system**

and then press **RETURN**. (Refer to the **SYSTEM** command in Chapter 2.)

## Definitions of Start Options

### BASIC *filename*

Specifies a program to be loaded and run immediately. If *filename* is present, GWBASIC proceeds as if a RUN *filename* command were typed. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows GWBASIC programs to be executed in batch mode using the BATCH facility of MS-DOS. Such programs should include a SYSTEM statement to return to MS-DOS when they have finished, allowing the next program in the batch stream to execute.

### BASIC <*stdin*

Redirects GWBASIC input from the file specified by *stdin*. When present, this entry must appear before any other options. See Redirection of Standard Input and Standard Output in this chapter.

### BASIC >*stdout*

Redirects GWBASIC output to the file specified by *stdout*. When present, this entry must appear before any other options. See Redirection of Standard Input and Standard Output in this chapter.

### BASIC /C:*combuffer size*

Sets the size of the buffer that is used for receiving data through a communication file. Your computer has a serial (RS-232C) port, so you may use this option on the GWBASIC command line.

/C:0      disables RS-232C support. Subsequent I/O attempts result in a Device Unavailable error.

/C:nnn    allocates *nnn* bytes for the communication buffers. GWBASIC allocates 128 bytes for the transmit buffer for each RS-232C port installed. If /C is omitted, GWBASIC allocates 256 bytes for receive and 128 bytes for transmit for each RS-232C port installed. GWBASIC ignores the /C option when RS-232C ports are not present.

### BASIC /D

If included in the command line, causes the Double Precision Transcendental math package to stay in memory. If omitted, the package is discarded, freeing memory for program use. ATP, COS, EXP, LOG, SIN, SQR, and TAN functions are converted to Double Precision.

### BASIC /F

Sets the maximum number of files that can be open at any one time when you are running a GWBASIC program. Each file needs 62 bytes of memory for the File Control Block (FCB), plus 128 bytes for a data buffer.

The size of the data buffer is set with the /S switch. If you don't enter /F on the command line, the default number of files is 3. The number of files that can be open at once depends on the value of the FILES command entered in CONFIG.SYS, the MS-DOS configuration file. The maximum number permissible for use with GWBASIC is FILES=15. If no FILES command is used in CONFIG.SYS, or if no CONFIG.SYS file is present, the default value for GWBASIC files is FILES=8.

The first three file handles are taken by *stdin*, *stdout*, *stderr*, *stdaux*, and *stdprn*. GWBASIC requires an additional handle for LOAD, NAME, CHAIN, MERGE, and SAVE. For example, if FILES=12, 8 are left for GWBASIC file I/O. If you attempt to OPEN a file after all the file handles have been exhausted, a Too many files error message is displayed.

BASIC /M:[*highest memory location*][,*maximum block size*]

Optionally sets the *highest memory location* that is used by GWBASIC. GWBASIC is able to use a maximum of 64K bytes of memory, so the highest value that may be set is 64K (FFFF in hexadecimal). If machine language subroutines are to be used with GWBASIC programs, use the /M option to set the highest memory location that GWBASIC can use. When the highest memory location is omitted or has a value of zero, GWBASIC tries to allocate all it can up to a maximum of 64 K.

If you want to load items into memory above the highest location GWBASIC can use, enter the optional parameter *maximum block size* to reserve space for them. *Maximum block size* must be in paragraphs. When omitted, &H1000 (4096) is assumed. This allocates 65536 bytes (65536 = 4096x16) for GWBASIC's Data and Stack segment. If you want 65536 bytes for GWBASIC and 512 bytes for machine language subroutines, use /M:&H1010 (4096 paragraphs for GWBASIC, 16 paragraphs for your routines). This option can also be used to limit the GWBASIC block in order to have more memory available. For example:

/M:2048

allocates 32728 bytes for data and stack.

/M:32000,2048

allocates 32728 maximum, but GWBASIC uses only the lower 32000. This leaves 768 bytes for your use.

BASIC /S:*buffer size*

Sets the buffer size for use with random files. The record length parameter on the OPEN statement may not exceed this value. The default buffer size is 128 bytes; the maximum value is 32767. A size of 512 generally improves performance when using open files.

**Note:** *Combuffer size*, *buffer size*, *highest memory location*, and *maximum block size* are numbers that maybe either decimal, octal (preceded by &O) or hexadecimal (preceded by &H) values.

## Examples

A>BASIC PAYROLL.BAS

Use all memory and load and execute PAYROLL.BAS.

A>BASIC /M:32768

Use only the first 32K of memory. 32K above that is free for the user.

A>BASIC DATAACK/M:&H9000

Use first 36K of memory and execute DATAACK.BAS.

A>BASIC /C:0/M:32768

Disable RS-232C support and use only the first 32K of memory. 32K above that is free for the user.

## Redirection of Standard Input and Standard Output

You can redirect GWBASIC's input and output to read from a Standard Input file and write to a Standard Output file by using the command line:

BASIC program name [*<input file*] [>] [*> output file*]

Both *input file* and *output file* can be a device instead of a disk file.

When redirecting Standard Input or Output, the following rules apply:

1. When redirected, all INPUT, LINE INPUT, INPUT\$, and INKEY\$ statements are read from the *input file* specified instead of from the keyboard.
2. All PRINT statements write to the *output file* specified instead of the screen.
3. Error messages go to Standard Output.
4. File input from "KYBD: " still reads from the keyboard.
5. File output to "SCRN: " still displays on your screen.
6. GWBASIC continues to trap keys from the keyboard when the ON KEY (n) statement is used.
7. The printer echo key does not cause LPT1 echoing if Standard Output has been redirected.
8. Pressing **CTRL BREAK** causes GWBASIC to close any open files, send the message BREAK in *linennnnn* to Standard Output.

9. When input is redirected, GWBASIC reads from its source until a **CTRL Z** is detected. This condition may be tested with the EOF function. If the file is not terminated with a **CTRL Z** or if a GWBASIC file input statement tries to read past end-of-file, then GWBASIC closes any open files, writes the message Read past end to Standard Output.

### Examples

BASIC SAMPLE >DATA.OUT

Data read by INPUT and LINE INPUT comes from the keyboard. Data output by PRINT is sent to the file DATA.OUT.

BASIC SAMPLE <DATA.IN

Data read by INPUT and LINE INPUT comes from the DATA.IN file. Data output by PRINT is displayed on the screen.

BASIC SAMPLE <INPUT.DAT >OUTPUT.DAT

Data read by INPUT and LINE INPUT comes from the file INPUT.DAT and data output by PRINT is sent to the file OUTPUT.DAT.

BASIC TRAVEL <\SALES\DIV\TRANS > >\SALES\SALES.DAT

Data read by INPUT and LINE INPUT now comes from the file \SALES\DIV\TRANS. Data output by PRINT is appended to the file named \SALES\SALES.DAT.

### Entering GWBASIC Commands

The Ok prompt on your screen means that GWBASIC is ready to accept any commands. Whenever the computer is in this *ready state*, you can create, change, or run a GWBASIC program. A *command mode* and a *program mode* are provided to execute GWBASIC commands and statements (see below, Modes of Operation).

In the command mode, each command entered is performed (executed) immediately. In the program mode, each command or statement entered is stored in memory before being executed. When the computer is in the ready state and a command or statement is entered, it is executed in command mode unless it is prefixed with a line number. If a command or statement has a line number, it is assumed to be a statement to be executed in the program mode.

Commands for displaying the contents of a program and handling files are usually executed in the command mode. This mode also enables immediate processing of numerical expressions, thus providing calculator-like features.

If you enter the `PRINT 5*6*100` command at the `Ok` prompt, for example, `GWBasic` executes it immediately in command mode:

```
Ok
PRINT 5*6*100
3000
Ok
```

When the `PRINT` statement above is entered as a command (i.e., with no line number), the numerical expression is immediately processed and the result of the calculation (in this case, 3000) is displayed. More complicated expressions such as those containing variables and functions can be computed in the same manner. Use program mode to perform repeated calculations or complex procedures.

The program mode is assumed when you enter a `RUN` command.

### How to Write a Program

The standard procedure for writing a program in `GWBasic` is:

1. Enter a `NEW` command, to clear the `GWBasic` program area in memory for entry of a new program.
2. Enter an `AUTO` command, to assign a line number automatically to each statement entered. Entry of this command may be omitted. If it is omitted, you have to assign numbers yourself. (See Chapter 2, `AUTO` for more information.)
3. Enter the statements of a program sequentially. A keying error can be corrected as follows:
  - o To erase the preceding character or characters, press **BACKSPACE**.
  - o To erase the entire line being entered, press **ESC**.
  - o To correct a line already entered, follow the program correction procedure explained in the next section.
4. When you have entered all the necessary statements, press **CTRL BREAK** to clear the `AUTO` command function.
5. Enter a `LIST` command to display the program just entered. Check each statement for typing errors.
6. Enter a `RUN` command to start the program.
7. To store the program in a disk file, enter the `SAVE` command, together with the name you wish to call the program. This name should appear in double quotes (" ").

Using the procedure described above, write a program to obtain the real root of a quadratic equation. For example:

```
Ok
NEW
Ok
AUTO
10 INPUT "COEFFICIENT A =";A
20 INPUT "COEFFICIENT B =";B
30 INPUT "COEFFICIENT C =";C
40 D=B^2-4*A*C
50 IF D<0 GOTO 110
60 X1=(-B+SQR(D))/(2*A)
70 X2=(-B-SQR(D))/(2*A)
80 PRINT "X1=";X1
90 PRINT "X2=";X2
100 STOP
110 PRINT "NO REAL ROOT"
120 END
```

Press **CTRL BREAK**.

```
Ok
RUN
COEFFICIENT A=?1
COEFFICIENT B=?2
COEFFICIENT C=?1
X1= -1
X2= -1
BREAK in 100
Ok
```

## How to Make Corrections to the Current Program

You can correct a GWBASIC program in one of two ways: *line correction* mode or *character correction* mode. This section describes line correction. Character correction is covered later in The Screen Editor section of this chapter.

In the line correction mode, correct a program as follows:

1. Enter a LIST command and check the contents of the program. If the program is not in memory, enter a LOAD command before the LIST command. The program is displayed in standard format.

To interrupt the listing, press **CTRL NUM LCK**. To restart the listing, press any key other than **CTRL NUM LCK**. To cancel the listing, press **CTRL BREAK**.

2. Add, insert, or replace a new line by assigning a line number to it. For example, to add a new line to a program which ends with line 100, assign line number 110. To insert a new line between lines 70 and 80, assign line number 75. To replace line 50, assign line number 50.



3. To delete a program line, enter only the line number of that line. To delete all the lines within a certain range, use a DELETE command as follows:

DELETE 600

Only line 600 is deleted.

DELETE 300-400

All lines between lines 300 and 400 are deleted.

DELETE -500

The lines up to line 500 are deleted.

DELETE 600-

Deletes line 600 and all higher lines.

4. Enter a LIST command again to check the corrected program. A LIST command can specify any of the following ranges:

LIST

The whole program is displayed.

LIST 300-400

Lines 300 to 400 are displayed.

LIST -500

The lines up to line 500 are displayed.

LIST 600-

The lines from line 600 are displayed.

## THE SCREEN EDITOR

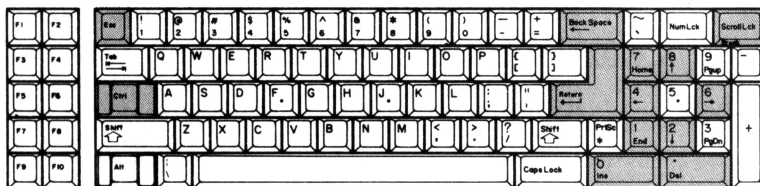
You can perform character corrections on an existing program using the screen editing feature. By operating the cursor control keys and edit keys, you can easily insert, add, or delete characters on the screen. See The Keyboard section in this chapter for more information.

### Screen Editor Corrections

1. Display the program line to be corrected using a LIST or an EDIT command. When an EDIT command is used, only the specified line is displayed and the cursor is positioned at the beginning of the line.
2. Use the cursor keys to move the cursor to the correct position.
3. Operate the keys to correct the character.
4. When the line is correct, press **RETURN**. This has the effect of entering the corrected line to replace the old line. If you do not press **RETURN**, the old line remains unchanged in memory. So, always press **RETURN** each time a correction is made.
5. Usually, when you press **RETURN**, GWBASIC accepts the line on the screen and processes it. You can use this to change the line number of the displayed line to copy it to a new line number position.

### Screen Editing Keys and Their Functions

The entire keyboard is discussed in detail in The Keyboard section of this chapter. The keys used for screen editing in GWBASIC are shown below:



**Figure 1-1** Model D Keyboard with Screen Editing Keys Shaded

The keys on the Model D's *numeric keypad* move the cursor to a position on the display screen. These keys, used in conjunction with **CTRL** and **END**, also perform editing functions on displayed program text. Table 1-1 lists the screen editing keys and their functions.

Key	Function
↑	Moves the cursor up one line.
↓	Moves the cursor down one line.
←	Moves the cursor left one position.
→	Moves the cursor right one position.
<b>CTRL</b>	Moves the cursor right to the next word.
<b>CTRL</b>	Moves the cursor left to the previous word.
<b>END</b>	Moves the cursor to the end of the current line.
<b>CTRL END</b>	Erases to the end of the current line.
<b>HOME</b>	Moves the cursor to the upper left corner of the screen.
<b>CTRL HOME</b>	Clears the screen and moves the cursor to the upper left corner.
<b>INS</b>	Sets insert mode ON or OFF.
<b>DEL</b>	Deletes the character at the current cursor location.
<b>BACKSPACE</b>	Deletes the last character typed (the character behind the cursor).
<b>ESC</b>	Erases the entire line where the cursor is located.
<b>CTRL BREAK</b>	Returns to GWBASIC command level without saving any editing changes made on the screen.

**Table 1-1** GWBASIC Screen Editing Keys

### **Modes of Operation**

When you start GWBASIC, the Ok prompt appears on the screen. Ok means GWBASIC is at command level, that is, it is ready to accept commands. At this point, GWBASIC may be used in either command (direct) mode or program (indirect) mode.

In command mode, GWBASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using GWBASIC as a calculator for quick computations that do not require a complete program.

Program mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

## LINE FORMAT

Program lines in a GWBASIC program have the following syntax (square brackets indicate optional entries):

line number GWBASIC STATEMENT[:GWBASIC STATEMENT]...]   **RETURN**

At your option, you can place more than one GWBASIC statement on a line, but each statement on a line must be separated from the last by a colon (:). The ellipsis (...) indicates that you can repeat the previous parameter.

A GWBASIC program line always begins with a *line number* and ends with a carriage return. Each line may contain a maximum of 255 characters or spaces, but most users use a maximum of 80 characters to prevent scrolling the screen.

## Line Numbers

Every GWBASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory. They are also used as references when branching and editing.

Line numbers must be in the range 0 to 65529.

In GWBASIC, a period (.) may be used in the EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

## USING CHARACTERS

### The Character Set

The GWBASIC character set comprises *alphabetic characters* (A-Z, upper and lower case), *numeric characters* (0-9), and a set of *special characters* (see Table 1-2).

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right Parenthesis
%	Percent sign
#	Number (or pound) sign
\$	Dollar sign
!	Exclamation point
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than sign
>	Greater than sign
\	Backslash or integer division symbol
"	Double quotation mark
_	Underscore

**Table 1-2** GWBASIC Special Characters

## Reserved Words

Certain words or abbreviations have a special meaning in GWBASIC. These are called *reserved words*. Reserved words include all of GWBASIC's commands, statements, function names, and operator names. These words cannot be used as variable names, although you may use a reserved word as *part* of a variable name.

Reserved words are always separated from other GWBASIC syntax units using *delimiters* such as commas or spaces. Reserved words are:

ABS	CALL	CLOSE
AND	CDBL	CLS
ASC	CHAIN	COLOR
ATN	CHDIR	COM
AUTO	CHR\$	COMMON
BEEP	CINT	CONT
BLOAD	CIRCLE	COS
BSAVE	CLEAR	CSNG

CSRLIN	KEY\$	REM
CVD	KILL	RENUM
CVI	LCOPY	RESET
CVS	LEFT\$	RESTORE
DATA	LEN	RESUME
DATE\$	LET	RETURN
DEF	LINE	RIGHT\$
DEFDBL	LIST	RMDIR
DEF FN	LLIST	RND
DEFINT	LOAD	RSET
DEFSNG	LOC	RUN
DEFSTR	LOCATE	SAVE
DELETE	LOF	SCREEN
DIM	LOG	SEG
DRAW	LPOS	SGN
EDIT	LPRINT	SHELL
ELSE	LSET	SIN
END	MERGE	SOUND
ENVIRON	MID\$	SPACE\$
ENVIRON	MKDIR	SPC
EOF	MKD\$	SQR
EQV	MKI\$	STEP
ERASE	MKS\$	STICK
ERDEV	MOD	STOP
ERDEV\$	MOTOR	STRIG
ERL	NAME	STR\$
ERR	NEW	STRING\$
ERROR	NEXT	SWAP
EXP	NOT	SYSTEM
FIELD	OCT\$	TAB
FILES	OFF	TAN
FIX	ON	THEN
FNxxxxx	OPEN	TIME\$
FOR	OPTION	TIMER
FRE	OR	TO
GET	OUT	TRON
GOSUB	PAINT	TROFF
GOTO	PALETTE	USING
HEX\$	PEEK	USR
IF	PEN	VAL
IMP	PLAY	VARPTR
INKEY\$	PMAP	VARPTR\$
INP	POINT	VIEW
INPUT	POKE	WAIT
INPUT#	POS	WEND
INPUT\$	PRESET	WHILE
INSTR	PRINT	WIDTH
INTER\$	PRINT\$	WINDOW
IOCTL	PSET	WRITE
IOCTL\$	PUT	WRITE#
INT	RANDOMIZE	XOR
KEY	READ	

## CONSTANTS

Constants are the actual values GWBASIC uses during execution. There are two types of constants: string and numeric.

A *string constant* is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants follow:

```
"HELLO"  
"$25,000.00"  
"Number of Employees"
```

*Numeric constants* are positive or negative numbers. Numeric constants in GWBASIC cannot contain commas. There are five types of numeric constants.

1. Integer constants

Whole numbers between -32768 and +32767. Integer constants do not have decimal points.

2. Fixed point constants

Positive or negative real numbers (numbers that contain decimal points).

3. Floating point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10E-38 to 10E+38. Examples:

```
235.988E-7 = .0000235988  
2359E6 = 2359000000
```

Double precision floating point constants use the letter D instead of E. See Single and Double Precision Form for Numeric Constants in this chapter.

4. Hex constants

Hexadecimal numbers with the prefix &H. Examples:

```
&H76  
&H32F
```

5. Octal constants

Octal numbers with the prefix &O or &. Examples:

```
&O347  
&1234
```

## Single and Double Precision Form for Numeric Constants

In GWBASIC, numeric constants may be either *single precision* or *double precision* numbers.

A single precision constant is any numeric constant that has:

1. Seven or fewer digits, or
2. Exponential form using E, or
3. A trailing exclamation point (!).

A double precision constant is any numeric constant that has:

1. Eight or more digits, or
2. Exponential form using D, or
3. A trailing number sign (#).

Double precision numbers are stored with 17 digits of precision, and printed with up to 16 digits.

Examples of single and double precision constants are listed below:

Single Precision Constants	Double Precision Constants
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Table 1-3 Examples of Single and Double Precision Constants

## VARIABLES

*Variables* are names used to represent values that are used in a GWBASIC program. You may assign the value of a variable explicitly, or you may assign it as the result of calculations in the program.

Before a numeric variable is assigned a value, its value is assumed to be zero (0). A string variable is assumed to be null.



## Variable Names and Declaration Characters

GWBasic variable names may be any length, but only the first 40 characters are significant. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character in the variable name must be a letter. Special *type declaration characters* are also allowed -- see below.

A variable name may not be a reserved word. GWBasic allows embedded reserved words. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all GWBasic commands, statements, function names, and operator names.

Variables may represent either a numeric value or a *string*. String variable names are written with a dollar sign (\$) as the last character. For example: A\$ = "Sales Division". The dollar sign is a variable type declaration character, that is, it "declares" that the variable represents a string.

In GWBasic, numeric variable names may declare integer, single, or double precision values. The type declaration characters for these variable names are:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of GWBasic variable names are:

PI#	declares a double precision value
MINIMUM!	declares a single precision value
LIMIT%	declares an integer value
N\$	declares a string value
ABC	declares a single precision value

In GWBasic, there is a second method by which variable types may be declared. The GWBasic statements DEFINT, DEFSTR, DEFSNG, and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Chapter 2.

## Array Variables

An *array* is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer expression.

An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

## Space Requirements

Variables	Bytes
INTEGER	2
SINGLE PRECISION	4
DOUBLE PRECISION	8
Arrays	Bytes
INTEGER	2 per element
SINGLE PRECISION	4 per element
DOUBLE PRECISION	8 per element
Strings	
3 bytes overhead plus the present contents of the string.	

Table 1-4 Space Requirements: Variables, Arrays, Strings

## TYPE CONVERSION

When necessary, GWBASIC converts a numeric constant from one type to another, for example, from single to double precision. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number is stored as the type declared in the variable name. If a string variable is set equal to a numeric value or vice versa, a Type mismatch error occurs. Examples:

```
10 A% = 23.42
20 PRINT A%
RUN
23
```

or

```
10 A$ = 23.42
20 PRINT A$
RUN
Type mismatch in 10
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision. Examples:

```
10 D# = 6#/7
20 PRINT D#
RUN
.8571428571428571
```

The arithmetic was performed in double precision, and the result was returned in D# as a double precision value.

```
10 D = 6#/7
20 PRINT D
RUN
.8571429
```

The arithmetic was performed in double precision, and the result was returned to D (single precision variable), rounded and printed as a single precision value.

3. Logical operators (see next section) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an Overflow error occurs.
4. When a floating point value is converted to an integer, the fractional portion is rounded. Example:

```
10 C% = 55.88
20 PRINT C%
RUN
56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number are valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value is less than 6.3E-8 times the original single precision value. Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
2.04 2.039999961853027
```

## EXPRESSIONS AND OPERATORS

An expression may be a simple string or numeric constant, a variable, or it may combine constants and variables with operators to produce a single value. Operators perform mathematical or logical operations on values. The operators provided by GWBASIC may be divided into four categories:

- Arithmetic
- Relational
- Logical
- Functional

## Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
$\wedge$	Exponential	$X^Y$
$-$	Negation	$-X$
$*$	Multiplication	$X*Y$
$/$	Floating Point Division	$X/Y$
$+$	Addition	$X+Y$
$-$	Subtraction	$X-Y$

**Table 1-5** Arithmetic Operation with Sample Expression

To change the order in which operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained. Here are some sample algebraic expressions and their GWBASIC counterparts.

Algebraic Expression	GWBASIC Expression
$X+2Y$	$X+Y*2$
$\frac{X-Y}{Z}$	$X-Y/Z$
$\frac{XY}{Z}$	$X*Y/Z$
$\frac{X+Y}{Z}$	$(X+Y)/Z$
$(X^2)^Y$	$(X^2)^Y$
$X^{Y^Z}$	$X^(Y^Z)$
$X(-Y)$	$X*(-Y)$

**Table 1-6** Algebraic Expressions and GWBASIC Counterparts

**Integer Division and Modulo Arithmetic.** Two additional operators are available in GWBASIC: integer division and modulo arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed and the quotient is truncated to an integer. For example:

$$\begin{aligned}10 \backslash 4 &= 2 \\ 25.68 \backslash 6.99 &= 3\end{aligned}$$

The precedence of integer division is just after multiplication and floating point division.

Modulo arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

$$\begin{aligned}10.4 \text{ MOD } 4 &= 2 && (10/4=2 \text{ with a remainder } 2) \\ 25.68 \text{ MOD } 6.99 &= 5 && (26/7=3 \text{ with a remainder } 5)\end{aligned}$$

The precedence of modulo arithmetic is just after integer division.

**Overflow and Division by Zero.** If, during the evaluation of an expression, a division by zero is encountered, the Division by zero error message is displayed. Machine infinity with the sign of the numerator is supplied as the result of the division and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the Division by zero error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the Overflow error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

## Relational Operators

Relational operators are used to compare two values. The result of the comparison is either true (-1) or false (0). This result may then be used to make a decision regarding program flow. (See IF, Chapter 2.)

Operator	Relation Tested	Expression
=	Equality	$X = Y$
<>	Inequality	$X <> Y$
<	Less than	$X < Y$
>	Greater than	$X > Y$
<=	Less than or equal to	$X <= Y$
>=	Greater than or equal to	$X >= Y$

The equal sign is also used to assign a value to a variable. See Chapter 2, LET.

**Table 1-7** Operators, Relation Tested, Expressions

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

$$X + Y < (T-1)/Z$$

is true if the value of X plus Y is less than the value of T-1 divided by Z. Here are more examples:

```
IF SIN(X)<0 GOTO 1000
```

```
IF I MOD J <> 0 THEN K=K+1
```

## Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a result that is either true (not zero) or false (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT		
X	NOT X	
T	F	
F	T	
AND		
X	Y	X AND Y
T	T	T
T	F	F
F	T	F
F	F	F
OR		
X	Y	X OR Y
T	T	T
T	F	T
F	T	T
F	F	F
XOR		
X	Y	X XOR Y
T	T	F
T	F	T
F	T	T
F	F	F
IMP		
X	Y	X IMP Y
T	T	T
T	F	F
F	T	T
F	F	T
EQV		
X	Y	X EQV Y
T	T	T
T	F	F
F	T	F
F	F	T

**Table 1-8** Logical Operators

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see Chapter 2, IF). For example:

```
IF D<200 AND F<4 THEN 80
```

```
IF I>10 OR K<0 THEN 50
```

```
IF NOT P THEN 100
```

Logical operators work by converting their operands to 16-bit signed, two's complement integers in the range -32768 to +32767 (if the operands are not in this range, an error results). If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to mask all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to merge two bytes to create a particular binary value. The following examples demonstrate how the logical operators work.

Example	Explanations
63 AND 16 = 16	63 = binary 111111 and 16 = binary 100000, so 63 AND 16 = 16
15 AND 14 = 14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8 = 8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10 = 10	10 = binary 1010, so 1010 OR 1010=1010 (10)
-1 OR -2 = -1	-1 = binary 1111111111111111 and -2 = binary 111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1
NOT X = -(X + 1)	The two's complement of any integer is the bit complement plus one

**Table 1-9** How Logical Operators Work



## Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. GWBASIC has *intrinsic* functions that reside in the program, such as SQR (square root) or SIN (sine). All GWBASIC intrinsic functions are described in Chapter 3.

GWBASIC also allows *user defined* functions that are written by the programmer. See Chapter 2, DEF FN.

## String Operations

Strings may be concatenated using the plus sign (+). For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

=            <>            <            >            <=            >=

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTH "
B$ < "9/12/78" where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

## ERROR MESSAGES

If GWBASIC detects an error that causes program execution to terminate, an error message is printed. In GWBASIC, the entire error message is printed. For a complete list of GWBASIC error codes and error messages, see Appendix F.

## FILES

A file is a collection of information on an input/output storage medium. Each file of information is identified by a name that is used to specify the particular file to be accessed by a GWBASIC command or statement.

The *filespec* parameter, used in the description of commands in this and subsequent chapters, has the following syntax:

[*d:*] [*path*] *filename* [*.ext*]

The *d:* indicates the drive where the required file is stored. You type in the letter of the floppy- or fixed-disk drive that contains the *filename* you specify. The colon (:) is required to separate the drive from *path* or *filename*. For example, C:FILE1.

*Path* indicates the path to follow in a directory structure. If *path* is specified, it must be separated from the *filename* by a final backslash (\) character. Usually, *path* is entered in the form \PATHNAME\, where the first backslash instructs GWBASIC to begin the path search in the root directory. For example, C:\DATA\FILE1. See the Directory Paths section in this chapter for additional information on paths.

Both *drive* and *path* are optional. If *drive* is omitted, the currently active disk drive is assumed. If *path* is omitted, the root directory is assumed.

*Filename* specifies any valid disk file. It is a character string consisting of 1 to 8 characters.

An optional *extension* may follow a *filename*. It consists of a period (.) followed by a character string of 3 characters or less. *Extension* usually indicates the file type.

If you specify a *filename* longer than 8 characters that does not contain an extension, GWBASIC terminates the *filename* at the 8th character, followed by a period and the subsequent 3 characters as the *extension*. If a *filename* contains an *extension* and is longer than 8 characters, only the first eight characters are retained.

If you specify an *extension* longer than 3 characters, GWBASIC terminates the *extension* after the first 3 characters.

When *extension* is omitted, .BAS is assumed. However, when it is omitted in a KILL, NAME, or OPEN statement, no extension (blank) is assumed.

Only the following characters are allowed for *filenames* and *extensions*:

Alphabetic	A through Z
Numerics	0 through 9
Special symbols	@ # \$ % ^ & ! - _ ' \ ~

## DEVICE NAMES

Table 1-10 below lists the devices recognized by GWBASIC and whether they are used as input devices, output devices, or both.

Device	Type of I/O Device	Input	Output
KYBD:	Keyboard	X	
SCRN:	Screen		X
LPT1:	Printer		X
LPT2:	Printer		X
LPT3:	Printer		X
COM1:	RS-232C unit number 0	X	X
COM2:	RS-232C unit number 1	X	X
COM3:	RS-232C unit number 2	X	X
A:	Disk Drive 1	X	X
B:	Disk Drive 2	X	X
C:	Fixed disk 1	X	X

Table 1-10 Devices

## MS-DOS FILE SYSTEM

Before Version 2.0, MS-DOS had a simple directory structure adequate for small (160KB to 320KB) disks. Now, with fixed disks capable of storing 5 to 20 (or more) million characters, the sheer number of directory entries becomes difficult to manage. Additionally, MS-DOS supports a wide variety of fixed file peripherals. The need for better disk organization is apparent.

People tend to think in hierarchical terms (organization charts and family trees, for example). It is possible to organize your files on disk in a similar manner.

As an example, in a business, both the sales and accounting departments share a computer with a large fixed-disk drive. The individual employees use it for preparation of reports and for maintaining accounting information. You could view the organization of files on the disk in this fashion:

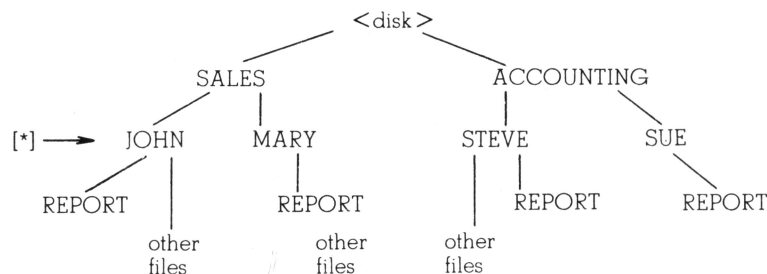


Figure 1-2 Hierarchical File Structure

## Directory Paths

A *directory path* is a series of directory names separated by the backslash character (\) and ending with a filename or a directory name. A path that starts at the root begins with a backslash (\).

There is a special directory entry in each directory, (denoted by ..) that is the parent of the directory. The root directory's parent is itself.

You can organize a disk so that the GWBASIC files you are not using in your current task do not interfere with the task.

To specify a filename, you can use one of two methods: either specify a path from the root node to the file, or specify a path from the current node to the file.

If you use a directory structure like the hierarchy shown in Figure 1-2 above, and assume that the current directory is at point [\*] (subdirectory JOHN), to reference the REPORT under JOHN, the following are all equivalent:

```
REPORT
\SALES\JOHN\REPORT
..\JOHN\REPORT
```

To refer to the REPORT under MARY, the following are all equivalent:

```
..\MARY\REPORT
\SALES\MARY\REPORT
```

To refer to the REPORT under SUE, the following are all equivalent:

```
..\..\ACCOUNTING\SUE\REPORT
\ACCOUNTING\SUE\REPORT
```

There are no restrictions on the depth of a tree (the length of the longest path from root to leaf) except in the number of allocation units available. The root directory may have a fixed number of entries. This number varies according to the size of the disk: from 64 for single-sided disks to 256 for a fixed disk. For non-root directories, there is no limit to the number of files per directory except in the number of allocation units available.

## USING THE SCREEN

GWBASIC allows drawing of text, special characters, points, lines, and more complex shapes in color or monochrome. Text refers to alphabetic characters, numbers, and special symbols contained in the character set.

## Display Types

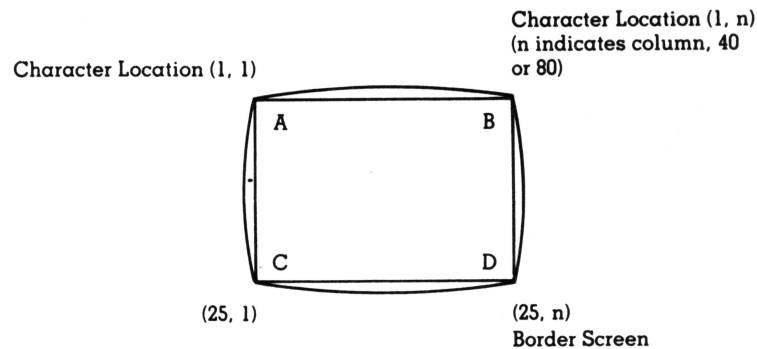
Two types of display units are available with the Model D: a monochrome display and a color display.

Text on the monochrome display appears on the screen in green or amber, with a black background. By setting a parameter in a COLOR statement, a blinking, reversed-image, invisible, highlighted, or underscored character can be displayed on the screen.

Text on the color display can be displayed in 16 colors.

The graphics mode allows drawing of complex pictures in many colors by addressing all pixels (dots) on the screen.

The screen layout is shown below:



**Figure 1-3** Screen Layout

The screen may be either 40 columns or 80 columns wide, specified by a WIDTH statement.

The character location may be specified by a LOCATE statement. Use of POS(0) and CSRLIN functions identifies the screen location of a character.

The area at the edge of the screen in which no characters may be displayed is called the border. The color of the border may also be specified with a COLOR statement.

The following statements are available for text processing in text mode:

CLS  
COLOR  
LOCATE  
PRINT  
SCREEN  
WIDTH  
WRITE

The following functions are provided:

CSRLIN  
POS  
SCREEN  
SPC  
TAB

When a color display is connected, each screen can access a set of pages: four pages in 80x25 mode and eight pages in 40x25 mode. In addition to the page to be displayed on the screen, the page to write to may be specified by a SCREEN statement.

## Graphics Mode

The following statements are available in graphics mode:

CIRCLE  
COLOR  
DRAW  
GET  
LINE  
PAINT  
PRESET  
PSET  
PUT  
SCREEN

The following function is available in graphics mode:

POINT

Four types of resolution are available in this mode. They are selected by the SCREEN statement:

Medium resolution	320x200 in four colors
High resolution	640x200 in two colors
High resolution	640x200 in sixteen colors
High resolution	720x348 in monochrome

1. Medium resolution (320x200 in four colors) allows 320 dots (horizontal) and 200 dots (vertical) to be identified. Medium resolution uses color codes 0, 1, 2, and 3. The color codes are associated with colors via the palette. Palette 0 and palette 1 are specified in a COLOR statement.

Text can be displayed in the graphics mode. Medium resolution has the same character size (40x25) as the text mode. The color codes for foreground and background are 3 and 0 respectively.

2. High resolution (640x200 in two colors) allows 640 dots horizontal and 200 dots vertical to be identified. Color 0 is black and color 1 is white.

High resolution (640x200 in sixteen colors) allows 640 dots horizontal and 200 dots vertical to be identified. Colors 0 through 15 are used, in accordance with the list under the COLOR (Text) statement.

High resolution (720x348 in monochrome) allows 720 dots horizontal and 348 dots vertical to be identified. Color 0 is black and any other color is white.

When text characters are displayed in high resolution, the character size is the same as for 80x25 in the text mode. In two color high resolution, the color codes for foreground and background are 1 and 0 (that is, white on black).

## Specification of Coordinates

To draw a dot by a graphics statement, positional data is required. Give data in the form of coordinates, generally in the form (x,y) where x indicates the horizontal position and y the vertical position. This is called *absolute form*, indicating an actual position on the screen.

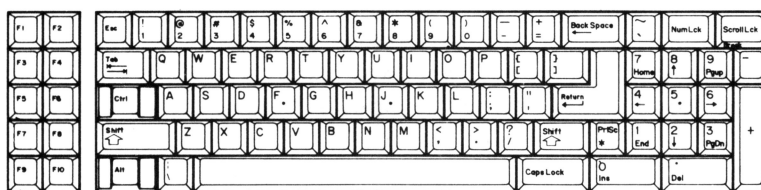
In addition, there is *relative form*, in which you specify an offset from the previous position referenced. The format is:

STEP(Xoffset, Yoffset)

Xoffset indicates the horizontal offset and Yoffset indicates the vertical offset.

The last referenced position depends on the execution result of each graphics statement.

## THE KEYBOARD



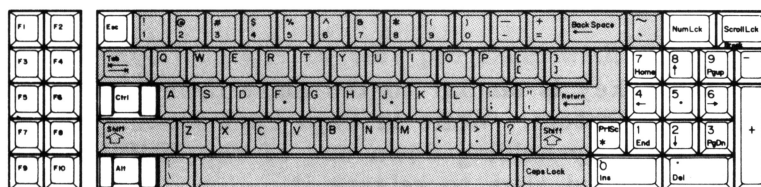
**Figure 1-4** Model D Keyboard

The keyboard of the Model D has three parts:

- o The familiar typewriter area, with letters and numbers, plus several special symbols, discussed below.
- o The ten function keys, positioned to the left of the typewriter area.
- o The numeric keypad, positioned to the right of the typewriter, that resembles a calculator.

### Typewriter Keyboard

The typewriter keyboard resembles a conventional typewriter, with letters and numbers in their customary places.



**Figure 1-5** Model D Keyboard with Conventional Keys Shaded

To display capital letters and the upper symbols on those keys with two characters, hold down either **SHIFT** key and press the desired key, just as you would on a typewriter.

Instead of a manual carriage return, there is a key labeled **RETURN**. Most entries you type are completed by pressing **RETURN**.



The Model D keyboard also has a number of special symbols and features not found on a conventional typewriter:

- o Symbols

Symbols like [, ], ~, and ^ are not usually found on standard typewriter keyboards, but have specific uses in computing.

- o Positioning

To accommodate some of the new keys, some standard keys are displaced in different positions. See, for example, the shifted period key, which has been replaced by the > symbol.

- o **BACKSPACE**

The **BACKSPACE** key erases as it moves from right to left. To move the cursor without erasing, use the (Left Arrow) key.



**Figure 1-6** Model D Keyboard with **ALT**, **BACKSPACE**, **CAPS LOCK**, **PRTSC**, and **CTRL** Keys Shaded

- o Uppercase

The **CAPS LOCK** key is similar to the Shift Lock key on a typewriter, but only capitalizes letters. While in Caps Lock mode, you can type lowercase letters by holding down one of the **SHIFT** keys. Press **CAPS LOCK** a second time to release it.

- o **PRTSC**

The **PRTSC** key (above \*) stands for Print Screen. To print a hard copy of what is being shown on the screen, press and hold one of the **SHIFT** keys, then press **PRTSC**.

## o ALT

The **ALT** (Alternate) key acts like a **SHIFT** key. If you press and hold it while pressing one of the following keys, it allows you to enter a GWBASIC statement with one keystroke.

A	AUTO	N	NEXT
B	BSAVE	O	OPEN
C	COLOR	P	PRINT
D	DELETE	Q	(no code)
E	ELSE	R	RUN
F	FOR	S	SCREEN
G	GOTO	T	THEN
H	HEX\$	U	USING
I	INPUT	V	VAL
J	(no code)	W	WIDTH
K	KEY	X	XOR
L	LOCATE	Y	(no code)
M	MERGE	Z	(no code)

You can enter characters not visible on the keys by holding the **ALT** key and typing the three digit ASCII code for the symbol you want, using the numeric keypad. See Appendix H, for a list of ASCII Character Codes.

## o CTRL

Like the **ALT** key, the **CTRL** key allows you to enter codes and characters not shown on the keyboard itself. Hold **CTRL** and press **G**, for example, and the computer beeps. This is called BEL (for Bell) and is an ASCII Control Code.

You may also use this key to edit programs with the program editor.

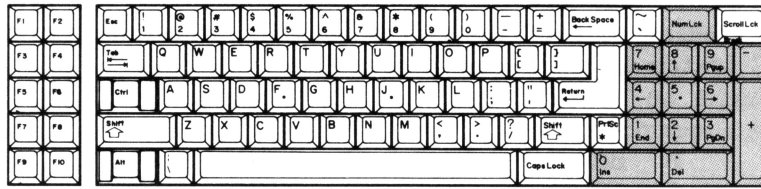
## Function Keys



**Figure 1-7** Model D Keyboard with Function Keys Shaded

The Function keys (**F1** to **F10**) are like the buttons on a car radio. You can set them to frequently-used commands or any desired sequence of characters. They can also be used as program interrupts in GWBASIC (see Chapter 2, ON KEY).

## Numeric Keypad



**Figure 1-8** Model D Keyboard with Numeric Keypad Shaded

This section of the keypad is used primarily for cursor movement editing functions. The Arrow, INS, and DEL keys move the cursor up, down, right, left, and permit insertion and deletion of characters.

You can use these keys like a calculator by pressing **NUM LCK**. Then, the number keys 0 through 9 and the decimal point on the **DEL** key work like a ten-key pad.

The **SCROLL LOCK**, **PGUP**, and **PGDN** keys are not used by GWBASIC, but may be used within a program.

### Special Key Combinations

The following combinations of key strokes perform specified functions:

- o **CTRL BREAK**

Interrupts program execution and returns to GWBASIC command level.  
Exits AUTO line numbering mode.

- o **CTRL NUM LCK**

Pauses the computer in the middle of a print or program process. To resume, press any other key except **CAPS LOCK** or **CTRL NUM LCK**.

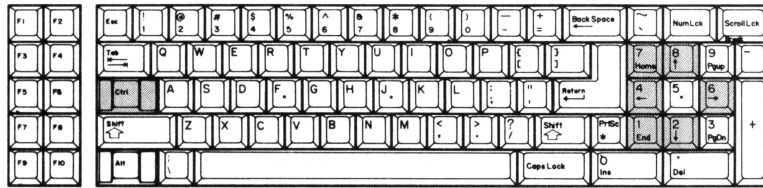
- o **CTRL ALT DEL**

Resets the system. Performs the same function as using the RESET button, or as switching the power OFF, then ON again, but faster. Hold **CTRL** and **ALT** down, then press **DEL**.

### GWBASIC PROGRAM EDITOR

When GWBASIC is at command level, any line typed is processed by the GWBASIC program editor. This is a screen line editor, which means that you can only change one line at a time anywhere on the screen. Any change you make is implemented by pressing **RETURN** after typing in the command.

## Special Program Editor Keys



**Figure 1-9** Highlighted CTRL, HOME, END, and Cursor Arrow Keys

Special program editing keys and their functions are described below:

- o     ↑     The Up Arrow moves the cursor one space up.
- o     ↓     The Down Arrow moves the cursor down one space.
- o     ←     The Left Arrow moves the cursor left one space. If the cursor moves beyond the far left edge of the screen, it will reappear at the right end of the preceding line.
- o     →     The Right Arrow moves the cursor right one space. If the cursor moves beyond the far right edge of the screen, it will reappear at the left end of the following line.
- o     **CTRL** →     This combination of keys pressed simultaneously moves the cursor to the next word. A "word" is any character or group of characters beginning with a letter or number separated by blanks or special character. For example:  
  

**JOHN, 17 MARK/OBEGE**

contains four words. The first letter of each word is shown in boldface for emphasis.
- o     **CTRL** ←     These keys move the cursor to the left of the previous word.
- o     **HOME**     Moves the cursor to the upper left corner of the screen.
- o     **CTRL HOME**     Clears the screen and moves the cursor one space up.
- o     **END**     Moves the cursor to the end of the logical line. Characters typed from this position are added to the end of the line.

**Note:** A logical line is a string of text that GWBASIC treats as a unit. A logical line may extend over more than one physical screen line by typing more characters than can be displayed on a single line. The cursor wraps around to the next screen line.

- o **CTRL END** Erase to the end of the logical line from the present cursor position. All screen lines are erased until the terminating carriage return is found.



**Figure 1-10** Highlighted CTRL, INS, DEL, BACKSPACE, ESC, BREAK, and TAB Keys

- o **INS** Sets insert mode to ON. If already ON, it turns insert mode OFF. When insert mode is OFF, any character typed replace existing characters on a line. When it is ON, existing characters are pushed to the right and to succeeding lines as new ones are inserted.
- o **DEL** Deletes the character at the present cursor position, and moves characters from right to left to fill in the blank created by the deletion.
- o **BACKSPACE** Deletes the last character typed (the character to the left of the cursor).
- o **ESC** Erases the entire line, when pressed anywhere in the line. The line is not passed to GWBASIC for processing. If it is a program line, it is not erased from the program in memory.
- o **CTRL BREAK** Returns to command level. Changes made to the current line being edited are not saved.
- o **TAB** When the insert mode is OFF, pressing TAB moves the cursor to the next tab stop without tampering with the text. When the insert mode is ON, pressing the tab key replaces the characters from the current cursor position to the next tab stop with blanks. There are tab stops every eight characters, at screen position 1, 9, 17, 25, and so on.

## **CHAPTER 2**

### **GW BASIC COMMANDS AND STATEMENTS**

All of the GW BASIC commands and statements are described in this chapter. Each description is formatted as follows:

#### **Purpose**

Explains the use of the command or statement.

#### **Syntax**

Shows the correct format for the command or statement.

#### **Comments**

Describes in detail how the command or statement is used.

#### **Examples**

Shows sample programs or program segments that demonstrate the use of the command or statement.

#### **Reference**

Lists other sections or GW BASIC commands you should read about for additional related information.

## AUTO

### Purpose

Generates a line number automatically after every carriage return.

### Syntax

AUTO [*line number* [, *increment* ]]

### Comments

AUTO begins numbering at *line number* and increments each subsequent *line number* by increment. The default for both values is 10. If *line number* is followed by a comma but increment is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk (\*) is printed after the number to warn the user that any input replaces the existing line. However, pressing RETURN immediately after the asterisk saves the line and generates the next line number.

AUTO is terminated by pressing CTRL BREAK. The line where CTRL BREAK is typed is not saved. After CTRL BREAK is pressed, GWBASIC returns to command level.

### Examples

AUTO 100,50

Generates line numbers 100, 150, 200...

AUTO

Generates line numbers 10, 20, 30, 40...

## **BEEP**

### **Purpose**

Sounds the speaker at approximately 1000 Hz for 1/4 second.

### **Syntax**

BEEP

### **Comments**

Both BEEP and PRINT CHR\$(7) have the same effect. In GWBASIC use PRINT CHR\$(7).

### **Example**

```
100 FOR I=1 to 5
110 BEEP
120 FOR J=0 TO 100:NEXT J
130 NEXT I
140 END
```



## BLOAD

### Purpose

Allows a file to be loaded anywhere in user memory.

### Syntax

BLOAD *filespec* [, *offset* ]

### Comments

*Filespec* is a valid string expression containing the device and filename. If the device name is omitted, the device in current use is assumed. When *filespec* does not agree with the rules concerning filenames, a Bad file name error is displayed and loading is discontinued.

*Offset* is a valid numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG statement. If *offset* is omitted, the offset specified at BSAVE is assumed. That is, the file is loaded into the same location it was saved from. If *offset* is specified, a DEF SEG statement should be executed before the BLOAD. When *offset* is used, GWBASIC assumes you want to BLOAD at an address other than the one saved.

This statement loads the memory image file (assembly language program, screen data, etc.) as specified in the *filespec* into user memory. BLOAD and BSAVE are most useful for loading and saving machine language programs (see CALL). However, use of BLOAD and BSAVE is not restricted to only machine language programs. Any segment may be specified as the source of target for these statements using the DEF SEG statement.

---

### WARNING

BLOAD does not perform an address range check. That is, it is possible to BLOAD anywhere in memory. You must not BLOAD over GWBASIC's stack, BASIC Program or GWBASIC's variable area.

---

### Example

```
100 'Load V-RAM data from "SCR"  
110 DEF SEG=&HB800  
120 BLOAD "SCR",0
```

### Reference

CALL, BSAVE

## BSAVE

### Purpose

Allows portions of the users memory to be saved on the specified device.

### Syntax

BSAVE *filespec* , *offset* , *length*

### Comments

*Filespec* is a valid string expression containing the device, filename, and extension. A path may be specified if necessary.

*Offset* is a valid numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG to start saving from.

*Length* is a valid numeric expression returning an unsigned integer in the range 1 to 65535. This is the length of the memory image to be saved.

BLOAD and BSAVE are most useful for loading and saving assembly language programs (see CALL). However, BLOAD and BSAVE are not restricted to only machine language programs. Any segment may be specified as the source or target for these statements via the DEF SEG statement.

When using a disk drive, if the device is not specified, the one in current use is assumed.

If the filename specified is less than 1 character or greater than 8 characters in length, a Bad file name error is issued and the save is aborted.

If *offset* is omitted, a Syntax error is issued and the save is aborted. A DEF SEG statement should be executed before the BSAVE. The last known DEF SEG address is always used for the save.

If *length* is omitted, a Syntax error is issued and the save is aborted.

### Example

```
100 DEF SEG=&HB800
120 BSAVE "SCR",0,&H8000
```

### Reference

BLOAD, CALL

## CALL

### Purpose

The CALL statement is the recommended way of interfacing assembly language programs with GWBASIC.

### Syntax

CALL *variable name* [(*argument list* [, *argument list*]...)]

### Comments

*Variable name* contains the address that is the starting point in memory of the subroutine being called. *Argument list* contains the variables or constants, separated by commas, that are to be passed to the routine.

The CALL statement transfers program control to an assembly language subroutine at the memory location whose address is expressed by *variable name*. (See the USR function for another way of interfacing assembly language programs with GWBASIC.)

**Note:** A variation of the CALL statement is CALLS. CALLS is similar to CALL, except that it passes segmented addresses of all arguments, whereas CALL passes unsegmented addresses. CALLS should be used when accessing routines written with the FORTRAN calling convention. All FORTRAN parameters are call-by-reference segmented address.

CALLS uses the segment address defined by the most recently executed DEF SEG statement to locate the routine being called.

### Example

```
100 DEF SEG=&H800
110 BLOAD "MACHINE EXE",0      'Load machine routine
.
.
.
300 SUBO=0
310 CALL SUBO(A,B,C)
```

### Reference

DEF SEG, DEF USR  
Appendix C

## CHAIN

### Purpose

Calls a program and passes variables to it from the current program.

### Syntax

```
CHAIN [ MERGE ] filespec [, [line number exp] [, ALL] [, DELETE range ]]
```

### Comments

*Filespec* is the name of the program that is called, in the format `[d:][ path ] filename [. ext ]`. For example:

```
CHAIN"PROD1"
```

*Line number exp* is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

**Note:** The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting. If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

### Examples

```
CHAIN"PROG1",1000
```

*Line number exp* is not affected by a RENUM command.

Every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed (see COMMON below). For example:

```
CHAIN"PROG1", 1000, ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the GWBASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGE<sub>d</sub>. For example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay can be brought in. To do this, use the DELETE option. For example:

```
CHAIN MERGE"OVRLAY2",1000,DELETE 1000-5000
```

The line numbers in range are affected by the RENUM command.

## CHDIR

### Purpose

Changes the current directory.

### Syntax

CHDIR *pathname*

### Comments

*Pathname* is a string expression not exceeding 128 characters that identifies the directory which is to be the current directory. CHDIR works exactly like the MS-DOS command CHDIR.

**Note:** You can create subdirectories within directories as in MS-DOS. Use the MKDIR command format. Remember to enclose the subdirectory names in quotation marks.

### Examples

Refer to the sample directory shown in Chapter 1, Figure 1-2. Assuming that your current directory is the root directory, make SALES the current directory on Drive A, and ACCOUNTING the current directory on Drive B:

```
CHDIR "SALES"
```

SALES is now the current directory on Drive A.

```
CHDIR "B:ACCOUNTING"
```

ACCOUNTING is now the current directory on Drive B.

Possible errors which may occur with the CHDIR command are:

- Bad filename
- Path not found
- Path/File Access error

Subsequent usage of *A:filespec* refers to files in the directory SALES.  
Usage of *B:filespec* refers to files in the B:ACCOUNTING directory.

## CIRCLE

### Purpose

Draws a circle with a center and radius as indicated by the first of its arguments.

### Syntax

CIRCLE (*xcenter*,*ycenter*),*radius* [, *color* [, *start* , *end* [, *aspect* ]]]

### Comments

The *xcenter* and *ycenter* coordinates can be specified in either absolute or relative form. For *color* display, refer to Using the Screen, Graphics Mode in Chapter 1.

*Radius* is the major axis of the circle, expressed in points. If the circle drawn is too large, parts of it may run off the screen.

*Color* specifies the color of the line segment. For the relationship between *color* and the actual colors, see COLOR below. The range of color depends on the graphics mode as follows:

Monitor	Mode	Range of Color	Default Value
Color	320x200	0 through 3	3
Color	640x200	0,1	1
Color	640x200	0 through 15	7
Monochrome	720x348	0,1	1

The *start* and *end* angle parameters are radian arguments between  $-2\pi$  and  $2\pi$ , which allows you to specify where the drawing of the circle begins and ends. If the *start* or *end* angle is negative (0 is not permitted), the circle is connected to the center point with a line, and the angles are treated as if they were positive (this is different from adding  $2\pi$ ).

*Aspect* ratio refers to the ratio of x radius to y radius. The default values are 5/6 for the 320 x 200 mode, 5/12 for the 640 x 200 mode, and 2/3 for the 720 x 348 mode. If the *aspect* ratio is less than 1, the radius is given in x-pixels. If it is greater than 1, the radius is given in y-pixels. The standard relative notation may be used to specify the center point.

## **CLEAR**

### **Purpose**

Sets all numeric variables to zero, sets all string variables to null, and closes all open files. Optionally, sets the end of memory and the amount of stack space.

### **Syntax**

```
CLEAR [[ expression1 ][, expression2 ]]
```

### **Comments**

*Expression1* is a memory location which, if specified, sets the highest location available for use by GWBASIC.

*Expression2* sets aside stack space for GWBASIC. The default is 512 bytes or one-eighth of the available memory, whichever is smaller.

### **Examples**

```
CLEAR  
CLEAR ,32768  
CLEAR ,,2000  
CLEAR 32768,2000
```



## CLOSE

### Purpose

Concludes I/O to a disk file.

### Syntax

CLOSE [[ # ] *file number* [, [ # ] *file number* ]...]

### Comments

*File number* is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and *file number* terminates upon execution of CLOSE. The file may then be reOPENed using the same or a different *file number*. Likewise, that *file number* may now be reused to OPEN any file. A CLOSE statement for a sequential output file writes the final buffer of output. The END, NEW, RESET, SYSTEM, or RUN commands without the R option always close all disk files automatically (STOP does not close disk files).

### Examples

See Appendix A.

## **CLS**

### **Purpose**

Erases the current active screen page.

### **Syntax**

CLS

### **Comments**

The CLS statement returns the cursor to home position in the upper left corner of the screen.

The SCREEN statement forces a screen clear if the resultant screen mode created is different than the mode currently in force.

The screen may also be cleared by pressing **CTRL HOME**.

### **Reference**

SCREEN

## COLOR (Text)

### Purpose

Sets the foreground, background, and border colors on the text screen.

### Syntax

COLOR [[ *foreground* ][, *background* ][, *border* ]]

### Comments

SCREEN 0:

*Foreground* is a numeric expression containing a value 0 to 31. It specifies the color of the characters.

*Background* is a numeric expression containing a value 0 to 7. It specifies the color of the background.

*Border* is a numeric expression containing a value 0 to 15. It specifies the color of the border screen.

When a color display is connected, the following colors are available for *foreground*:

0	Black	8	Gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Yellow
7	White	15	High-intensity white

When 16 is added to the above color codes, the character starts blinking.

Of the above color codes 0 through 7, only one color is used for *background*.

SCREEN 3:

*Foreground* is a numeric expression containing a value 0 to 15. It specifies the color of the characters.

*Background* is a numeric expression containing a value 0 to 15. It specifies the color of the background.

*Foreground* and *background* can not have the same values.

*Border* can not be specified using SCREEN 3.

#### MONOCHROME DISPLAY:

When a monochrome display is connected, the following colors are available for *foreground*:

0	Black
1	Underscored white
2-7	White

When using a color display, if you add 8 to a color code shown above, that character is highlighted. For example, 10 is high-intensity white. (Note that there is no high-intensity black.) When 16 is added, the character starts blinking.

For example, 16 is a blinking black character and 31 is a blinking high-intensity white character.

When a monochrome display is connected, the following colors are available for *background*.

0-6	Black
7	White

**Note:** Each parameter can be omitted. If it is omitted, the old value remains. If the value of a parameter exceeds 31, an illegal function call error message is displayed. The old value remains. If a COLOR statement terminates with a comma, a Missing operand error is displayed, but the specified color change remains valid. For example, the following statement is illegal:

COLOR 5,

## COLOR (Graphics)

### Purpose

Sets the colors to be used in the graphics mode.

### Syntax

COLOR [*background* ][,*palette* ]]

### Comments

*Background* is a numeric expression to specify the color of the background. Values 0 through 15 are allowed for *background*. For the relationship between these values and the actual colors, see the COLOR (Text) statement above.

*Palette* is a numeric expression to select a color palette.

In SCREEN 1, the colors available for selection by each palette are:

Color	Palette 0	Palette 1
0	Background	Background
1	Green	Cyan
2	Red	Magenta
3	Brown	White

If a *palette* is an even number, palette 0 is selected; if it is an odd number, palette 1 is selected.

Each parameter can be omitted. If it is omitted, the old value remains.

If the value of a parameter exceeds 31, an illegal function call error occurs. In this case, the old value remains.

### Example

```
5 SCREEN 1
10 COLOR 9,1
```

## COM

### Purpose

Prepares for trapping from a communications port.

### Syntax

```
COM(port number) ON  
COM(port number) OFF  
COM(port number) STOP
```

### Comments

*Port number* is the communication port number, 1 or 2.

Incoming communications can either be trapped (ON), ignored (OFF), or stopped (STOP).

After COM(*port number*) ON is set, the ON COM statement traps all incoming communications from the specified port, handling them as specified in the ON COM statement.

When COM(*port number*) OFF is used, all incoming communications are not trapped and are ignored.

The COM(*port number*) STOP statement does not allow trapping but holds the incoming communications until the COM(*port number*) ON statement is used to allow trapping of the communications that have been STOPped.

## **COMMON**

### **Purpose**

Passes variables to a CHAINED program.

### **Syntax**

```
COMMON variable [[, variable ]...]
```

### **Comments**

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, although it is recommended that they appear at the beginning. The same *variable* cannot appear in more than one COMMON statement. Array variables are specified by appending ( ) to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

### **Example**

```
100 COMMON A,B,C,D( ),G$  
110 CHAIN "PROG3",10
```

## **CONT**

### **Purpose**

Continues program execution after **CTRL BREAK** has been pressed, or a **STOP** or **END** statement has been executed.

### **Syntax**

CONT

### **Comments**

The program resumes at the point where the break occurred. If the break occurred after a prompt from an **INPUT** statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with **STOP** for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or direct mode **GOTO**, which resumes execution at a specified line number. With **GWBASIC**, CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break.

### **Examples**

See **STOP** for examples.



## DATA

### Purpose

Stores the numeric and string constants that are accessed by the program's READ statement(s).

### Syntax

DATA *constant* [[, *constant* ]...]

### Comments

DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as fit on a line (separated by commas), and any number of DATA statements may be used in a program. See READ for additional information.

The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

*Constant* may contain numeric constants in any format: fixed point, floating point, or integer (no numeric expressions are allowed). String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding *constant* in the DATA statement.

DATA statements may be reread from the beginning of the list using the RESTORE statement.

### Examples

See READ for examples.

## DEF FN

### Purpose

Defines and names a function that is written by the user.

### Syntax

```
DEF FNname [(parameter list)] = function definition
```

### Comments

*Name* must be a legal variable name. This *name*, preceded by FN, becomes the name of the function. *Parameter list* comprises those variable names in the *function definition* that are to be replaced when the function is called. The items in the list are separated by commas.

*Function definition* is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name.

A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that are given in the function call.

In GWBASIC, user-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a Type mismatch error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an Undefined user function error occurs. DEF FN is illegal in the direct mode.

### Example

```
410 DEF FNAB(X,Y)=X^3/Y^2  
420 T=FBAB(I,J)
```

Line 410 defines the function FNAB. The function is called in line 420.

## DEFINT/SNG/DBL/STR

### Purpose

Declares variable types as integer, single precision, double precision, or string.

### Syntax

*DEFtype range(s) of letters*

where *type* is INT, SNG, DBL, or STR

### Comments

A *DEFtype* statement declares that the variable names beginning with the letter(s) specified are that type variable. However, a type declaration character always takes precedence over a *DEFtype* statement in the typing of a variable.

If no type declaration statements are encountered, GWBASIC assumes all variables without declaration characters are single precision variables.

### Examples

```
10 DEFDBL L-P
```

All variables beginning with the letters L, M, N, O, and P are double precision variables.

```
10 DEFSTR A
```

All variables beginning with the letter A are string variables.

```
10 DEFINT I-N,W-Z
```

All variables beginning with the letters I, J, K, L, M, N, W, X, Y, and Z are integer variables.

## DEF SEG

### Purpose

Defines the current segment of storage.

### Syntax

DEF SEG [= *address* ]

### Comments

*Address* is a valid numeric expression returning an unsigned integer in the range 0 to 65535.

The *address* specified is saved for use as the segment required by the BLOAD, BSAVE, PEEK, POKE, VARPTR, USR, and CALL statements.

If the *address* option is omitted, the segment to be used is set to GWBASIC's Data Segment. This is the initial default value.

If the *address* option is given, it should be a value based upon a 16-byte boundary.

For example, if *address* is set at &H800, the starting address becomes &H8000.

GWBASIC does not perform additional checking to assure that the resultant segment:offset value is valid.

DEF and SEG must be separated by a space. Otherwise, GWBASIC interprets the statement DEFSEG=100 to mean "assign the value 100 to the variable DEFSEG."

Any value entered outside of this range (0 to 65535) results in an illegal function call error. The previous value is retained.

### Example

```
100 PRINT
110 DEF SEG=&H800
120 DT=&H55
130 FOR AD=0 TO 32000 STEP 2
140 POKE AD,DT
150 NEXT AD
160 END
```

## DEF USR

### Purpose

Specifies the starting address of an assembly language subroutine.

### Syntax

DEF USR [ *digit* ] = *integer expression*

### Comments

*Digit* may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If *digit* is omitted, DEF USR0 is assumed. The value of *integer expression* is the starting address of the USR routine. See Appendix C, Assembly Language Subroutines, for more information.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

### Example

```
.  
.   
.   
200 DEF USR0=24000  
210 X=USR0(Y^2/2.89)  
.   
.   
. 
```

## **DELETE**

### **Purpose**

Deletes program lines.

### **Syntax**

DELETE [ *line number* ][- *line number* ]

### **Comments**

GWBASIC always returns to command level after DELETE is executed. If *line number* does not exist, an illegal function call error occurs.

### **Examples**

DELETE 40

Deletes Line 40.

DELETE 40-100

Deletes lines 40 through 100.

DELETE -40

Deletes all lines up to and including line 40.

## **DIM**

### **Purpose**

Specifies the maximum values for array variable subscripts and allocates storage accordingly.

### **Syntax**

*DIM list of subscripted variables*

### **Comments**

If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a Subscript out of range error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

### **Example**

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
```

## DRAW

### Purpose

Draws figures on the screen.

### Syntax

DRAW *command string*

### Comments

A GML command is a single character within a *command string*, optionally followed by one or more arguments.

A shape is defined using the *command string*. This shape is displayed when the DRAW statement is executed. GWBASIC interprets the *command string* letter by letter, and executes it. The GML command is composed of movement commands.

Each of the following movement commands begin movement from the current graphics position. This is usually the coordinate of the last graphics point plotted with LINE, PSET, or another GML command. The *n* in the following commands indicates the distance of movement. The actual points moved is *n* adjusted to the scale factor set by the S command.

Un	Move up
Dn	Move down
Ln	Move left
Rn	Move right
En	Move diagonally up and right
Fn	Move diagonally down and right
Gn	Move diagonally down and left
Hn	Move diagonally up and left

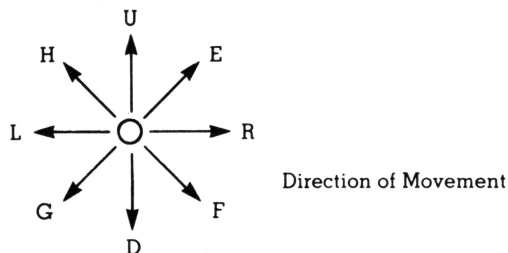


Figure 2-1 DRAW Commands -- Direction of Movement



**Mx,y**

Move absolute or relative. If *x* is preceded by a plus sign (+) or minus sign (-), *x* and *y* are added to the current graphics position and connected to the current position with a line. Otherwise, a line is drawn to point *x,y* from the current position.

The spacing of points in horizontal, vertical, and diagonal movements is influenced by the screen aspect ratio (screen width divided by the screen height).

The following prefix commands may precede any of the above movement commands.

**B**

Move but don't plot any points.

**N**

Move but return to original position when done.

**An:**

Sets angle *n*. *n* may range from 0 to 3 where 0 is 0 degrees, 1 is 90 degrees, 2 is 180 degrees, and 3 is 270 degrees. Figures rotated 90 or 270 degrees are scaled so that they appear the same size as with 0 or 180 degrees on a monitor with the standard aspect ratio of 4/3.

**Cn:**

Sets color *n*. The default is described in CIRCLE statement.

**Sn:**

Sets scale factor. *n* may range from 1 to 255 and 1/4 of this value becomes the scale factor. For example, if 1 is used, the scale factor is 1/4. The scale factor is multiplied by the distances given with U, D, L, R, E, F, G, H or relative M commands to get the actual distance traveled.

**Ppaint color,border color**

*Paint color* chooses an attribute from the attribute range for the current screen mode. The *border color* parameter sets the border color of the figure to be filled in the attribute range for the current screen mode. You must specify both *paint color* and *border color*.

**TAdegrees**

Rotate *degrees*. *Degrees* must be in the range -360 to 360 degrees. If *degrees* is positive, rotation is counterclockwise. If *degrees* is negative, rotation is clockwise.

### *Xstring*:

Executes a *substring* (not supported by GWBASIC compiler). This powerful command allows you to execute a second *substring* from a *string*, much like GOSUB in GWBASIC. You can have one *string* execute another, which executes a third, and so on.

Numeric arguments can be constants such as 123 or =variable:, where *variable* is the name of a variable.

The semicolon (;), besides being used with variables, must be written in the X command. In the standard commands defined in the argument, the semicolon can also be used to separate commands.

## Reference

### COLOR

## Examples

```
100 CLS
110 SCREEN 1
120 H=199:W=319
130 DRAW "BMO,0"
140 WHILE H>=1
150 DRAW "R=W;D=H;L=W;U=H;"
160 HO=H/8;WO=W/8
170 DRAW "BR=WO;BD=HO;"
180 H=H*3/4
190 W=W*3/4
200 WEND
210 END
```

DRAW "U50R50D50L50"	'Draw a box
DRAW "BE10"	'Move up and right into box
DRAW "P1,3"	'Paint interior

FOR D=0 TO 360	'Draw some spokes
DRAW "TA=D;NU50"	
NEXT D	

If you have a monochrome monitor, use SCREEN 3 instead of SCREEN 1 in line 110, and set the H and W values in line 120 to 347 and 719. If you have a color monitor and want to run the program in 16 color mode, use SCREEN 3 instead of SCREEN 1 in line 110, and set the H and W values in line 120 to 199 and 639.

## EDIT

### Purpose

Enters Edit Mode at the specified line.

### Syntax

```
EDIT line number  
EDIT .
```

### Comments

In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, GWBASIC types the line number of the line to be edited, then types a space for an Edit Mode subcommand.

*Line number* is the program line number of a line existing in the program. If there is no such line, the Undefined Line Number error message is displayed.

If you have just entered a line and wish to go back and edit it, entering a period (.) after the EDIT command enters Edit Mode at the current line. The line number symbol (.) always refers to the current line. This command can be used when an error message appears or immediately after a STOP command.

The LIST command is usually used to display the contents of a program.

**END**

**Purpose**

Terminates program execution, closes all files and returns to command level.

**Syntax**

END

**Comments**

END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. GWBASIC always returns to command level after an END is executed.

**Example**

```
520 IF K>1000 THEN END ELSE GOTO 20
```

## ENVIRON

### Purpose

Modifies a parameter in MS-DOS Environment String Table.

### Syntax

ENVIRON *string*

### Comments

*String* is a string expression. The value of the expression must be of the form *parameter-id=text*, or *parameter-id text*. Everything to the left of the equal sign or space is assumed to be a parameter, and everything to the right is assumed to be text.

If the parameter-id has not previously existed in the Environment String Table, it is appended to the end of the table. If the parameter-id exists on the table when ENVIRON is executed, the existing parameter-id is deleted and the new one is appended to the end of the table.

The text string is the new parameter text. If the text is a null string (" ") or is only a semicolon (;), then the existing parameter-id is removed from the table and the table is compressed.

ENVIRON can be used to change the PATH parameter for a child process, or to pass parameters to a child by inventing a new Environment Parameter (see MS-DOS PATH command).

Possible errors include parameters that are not strings and an Out of memory error when no more space can be given to the Environment String Table. The amount of free space in the table is usually very small.

### Examples

```
PATH=A:\
```

This MS-DOS command creates a default PATH to the root directory on Drive A:.

```
ENVIRON "PATH=A:SALES;A:ACCOUNTING"
```

Changes the PATH to a new value.

ENVIRON "SALES=PLAN"

Adds a new parameter to the Environment String Table. The Environment String Table now contains:

PATH=A:SALES;A:ACCOUNTING  
SALES=PLAN

By entering

ENVIRON "SALES=;"

the SALES entry is removed from the table.

### Reference

ENVIRON\$, SHELL

## **ERASE**

### **Purpose**

Eliminates arrays from a program.

### **Syntax**

ERASE *list of array variables*

### **Comments**

You can redimension arrays after they are erased, or use the previously allocated array space in memory for other purposes. If an attempt is made to redimension an array without first erasing it, a Redimensioned array error occurs.

### **Example**

```
.  
.   
.   
450 ERASE A,B  
450 DIM B(99)  
.   
.   
.
```

## ERROR

### Purpose

Simulates the occurrence of a GWBASIC error or allows error codes to be defined by the user.

### Syntax

ERROR *integer expression*

### Comments

The value of *integer expression* must be greater than 0 and less than 255. If the value of *integer expression* equals an error code already in use by GWBASIC (see Appendix F), the ERROR statement simulates the occurrence of that error, and the corresponding error message is printed. (See Example 1 below.)

To define your own error code, use a value that is greater than any used by GWBASIC error codes. It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to GWBASIC. This user-defined error code may then be conveniently handled in an error trap routine (see Example 2 below).

If an ERROR statement specifies a code for which no error message has been defined, GWBASIC responds with the message Unprintable error. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

### Examples

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in 30
```

Or, in direct mode:

```
Ok
ERROR 15                (You type this line)
String too long         (GWBASIC replies with this line)
Ok
```



```
.  
.   
.   
110 ON ERROR GOTO 400  
120 INPUT "WHAT IS YOUR BET";B  
130 IF B > 5000 THEN ERROR 210  
.   
.   
.   
400 IF ERR = 210 THEN PRINT  
    "HOUSE LIMIT IS $5000"  
410 IF ERL = 130 THEN RESUME 120  
.   
.   
. 
```

## FIELD

### Purpose

Allocates space for variables in a random file buffer.

### Syntax

FIELD [#] *file number*, *field width* AS *string variable*...

### Comments

To get data out of a random buffer after a GET, or to enter data before a PUT, you must execute a FIELD statement.

*File number* is the number under which the file was OPENed. *Field width* is the number of characters to be allocated to *string variable*. For example:

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N\$, the next 10 positions to ID\$, and the next 40 positions to ADD\$. FIELD does not place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length specified when the file was OPENed. Otherwise, a Field overflow error occurs (the default record length is 128).

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

**Note:** Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

### Examples

See Appendix A.

## FILES

### Purpose

Displays the names of files on the disk.

### Syntax

```
FILES [filespec]
```

### Comments

*Filespec* is a string expression used to specify the file.

When the *filespec* contains only the file name, all files using that name are displayed. When *filespec* is omitted, all the files on the current drive are listed. The filename specified may contain wildcard characters. Question marks (?) match any character in the filename or extension. An asterisk (\*) as the first character of the filename or extension matches any file or extension.

If the disk drive is specified in the *filespec*, the files on that drive are displayed. If the disk drive is not specified, the current drive is assumed.

### Examples

```
FILES "\SALES"
```

Displays the directory entry SALES<DIR>\.

```
FILES "\SALES\MARY\*.BAK"
```

Displays all backup (.BAK) files in subdirectory MARY.

## FOR...NEXT

### Purpose

Allows a series of instructions to be performed in a loop a given number of times.

### Syntax

```
FOR variable = x TO y [STEP z]
.
.
.
NEXT [variable ][[, variable ]...]
```

where x, y, and z are numeric expressions.

### Comments

*Variable* is used as a counter. The first numeric expression (x) is the initial value of the counter and the second numeric expression (y) is the final value of the counter. The third numeric expression (z) is used as an increment for STEP.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y).

If it is not greater, GWBASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

**Nested Loops.** FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique *variable* name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a separate NEXT statement must be used for each of them.

The *variables* in the NEXT statements may be omitted, in which case the NEXT statement matches the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a NEXT without FOR error message is issued and execution is terminated.

### Examples

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1 20
 3 30
 5 40
 7 50
 9 60
Ok
```

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. Previous versions of GWBASIC set the initial value of the loop variable before setting the final value; in that case, the above loop would have executed six times.

## GET

### Purpose

Reads a record from a random disk file into a random buffer and/or to get the permissible fixed length I/O for communication files.

### Syntax

```
GET [#] file number [, record number ]  
GET file number , n bytes
```

### Comments

*File number* is the number under which the file was OPENed. If *record number* is omitted, the next record (after the last GET) is read into the buffer. The largest possible *record number* is 32767.

*n bytes* is an integer expression returning the number of bytes to be transferred into or out of the file buffer.

**Note:** After a GET statement, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer.

### Examples

See Appendix A.

## GET (Graphics)

### Purpose

Reads points from an area of the screen.

### Syntax

```
GET array name  
GET (x1,y1)-(x2,y2), array name
```

### Comments

(x1,y1) and (x2,y2) are the coordinates of the opposite corners of the screen area.

GET reads the colors of the points in the specified screen area into the array. A rectangular area of the screen has as its opposite corners points (x1,y1) and (x2,y2). If you use the B option in the LINE statement, the results are the same.

This array is only to hold the image in the specified area without concern as to precision, but it must be in numeric form. Size is calculated as follows:

$$\text{array size} = (4 + \text{INT}((X * M + 7) / 8) * Z * Y) / A$$

where

- X Is the number of horizontal dots.
- Y Is the number of vertical dots.
- A Is the number of bytes in the array element.
- M\*Z Is the number of bits necessary to represent one dot where M and Z are from the following table.

Graphics Mode	M	Z
SCREEN 1	2	1
SCREEN 2	1	1
SCREEN 3	1	4
SCREEN 3 MONO	1	1

For example, the number of bytes required to make a 15 by 20 image in elementary medium resolution would be calculated as:

$$4 + \text{INT}((15 * 2 + 7) / 8) * 20 * 1$$

or, in other words, 84. The number of bytes per array element depends on the precision used when dimensioning the array. The following rules apply:

- 2 bytes for an integer
- 4 bytes for single precision
- 8 bytes for double precision

An integer array would thus have 42 elements. Storage of screen information using GET is:

1. 2 bytes giving the x dimension in bits
2. 2 bytes giving the y dimension in bits
3. The data itself.

If an integer array is used, it is possible to examine the x and y dimensions. The x dimension is in element 0 and the y dimension is in element 1 of the array. The integers are stored low byte first, but the data is transferred high byte first.

The data for each row of points in the rectangle is left justified on a byte boundary. If the number of bits stored is less than a multiple of 8, the rest of the byte is filled with zeros.

### Example

```
100 SCREEN 1,0; COLOR 0,0
110 FOR I=1 TO 10
120 LINE (0,0)-(RND*319,RND*199),RND*2+1
130 NEXT
140 FOR I=1 to 10
150 LINE (319,199)-(RND*319,RND*199),RND*2+1
160 NEXT
170 DIM A%(128)
180 GET (100,100)-(110,120),A%
190 END
```

### Reference

PUT



## GOSUB...RETURN

### Purpose

Branches to and returns from a subroutine.

### Syntax

```
GOSUB line number
.
.
.
RETURN
```

### Comments

*Line number* is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause GWBASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

### Example

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

## GOTO

### Purpose

Branches unconditionally out of the normal program sequence to a specific line number.

### Syntax

GOTO *line number*

### Comments

If *line number* is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after *line number*.

### Example

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5
AREA = 78.5
R = 7
AREA = 153.86
R = 12
AREA = 452.16
  Out of data in 10
Ok
```

## IF

### Purpose

Makes a decision regarding program flow based on the result returned by an expression.

### Syntax

```
IF expression THEN statement(s) line number  
[ELSE statement(s) line number ]
```

```
IF expression GOTO line number  
[ELSE statement(s) line number ]
```

### Comments

If the result of *expression* is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a *line number* for branching or one or more *statements* to be executed. GOTO is always followed by a *line number*.

If the result of *expression* is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement.

**Nesting of IF Statements.** The GWBASIC IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example:

```
IF X > Y THEN PRINT "GREATER" ELSE IF Y > X  
    THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

```
IF A = B THEN IF B = C THEN PRINT "A = C"  
    ELSE PRINT "A <> C"
```

If an IF...THEN statement is followed by a *line number* in the direct mode, an Undefined line error results unless a *statement* with the specified *line number* had previously been entered in the indirect mode.

**Note:** When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

### Examples

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if I is not zero.

```
100 IF(I<20)*(I>10) THEN DB = 1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
```

```
.
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of the IOFLAG variable. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the terminal.

## INPUT

### Purpose

Allows input from the terminal during program execution.

### Syntax

```
INPUT [;]["prompt string"];[[variable][,variable]...]
```

### Comments

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If "*prompt string*" is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement

```
INPUT "ENTER BIRTHDATE";B$
```

prints the prompt with no question mark.

If INPUT is immediately followed by a semicolon, the carriage return you typed to input data does not echo a carriage return/line feed sequence.

The data entered is assigned to the *variables* specified. The number of data items supplied must be the same as the number of *variables* in the list. Data items are separated by commas.

The *variable* names in the list may be numeric or string variables (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. Strings input to an INPUT statement need not be surrounded by quotation marks.

Responding to INPUT with too many or too few items, or with the wrong type of a value (for example, numeric instead of string), causes the message ?Redo from start to be printed. No assignment of input values is made until an acceptable response is given.

## Examples

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
? 5
```

(The 5 was typed in by the user  
in response to the question mark.)

```
5 SQUARED IS 25
Ok
```

```
LIST
10 PI = 3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A = PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
```

```
RUN
```

```
WHAT IS THE RADIUS? 7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS 171.9464
```

```
WHAT IS THE RADIUS?
etc.
```

## **INPUT#**

### **Purpose**

Reads data items from a sequential disk file and assigns them to program variables.

### **Syntax**

`INPUT# file number,variable [[,variable ]...]`

### **Comments**

*File number* is the number used when the file was OPENed for input. *Variables* are assigned to the items in the file. The variable type must match the type specified by the variable name. With INPUT#, unlike INPUT, no question mark is printed.

The data items in the file should appear just as they would if you typed data in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feed are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

If GWBASIC is scanning the sequential data file for a string item, then leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ( " ), the string item consists of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and terminates on a comma, carriage return, or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

### **Examples**

See Appendix B.

## IOCTL

### Purpose

Transmits a control character or string to a device driver.

### Syntax

IOCTL [#] *file number, string*

### Comments

IOCTL commands are generally two or three characters followed by an optional alphanumeric argument. An IOCTL command string may be up to 255 bytes long.

IOCTL works only if:

1. The device driver is installed.
2. The device driver states it processes IOCTL strings.
3. GWBASIC performs an OPEN command on a file on that device.

**Note:** Most standard MS-DOS device drivers do not process IOCTL strings. You must determine whether a specific driver can handle this command.

### Example

```
10 OPEN "\DEV\LPT1" FOR OUTPUT AS #1
20 IOCTL #1, "PL66"
```

Sets the page length to 66 lines per page on LPT1:.

### Reference

IOCTL\$



## KEY

### Purpose

Defines and/or displays the function key assignment text.

### Syntax

KEY *key number*, *string expression*  
KEY LIST  
KEY ON  
KEY OFF

### Comments

The KEY statement allows function keys to be designated as "soft" (reprogrammable) keys. *Key number* is the number of the function key (F1-F10, entered as 1-10). *String expression* is the text of a sequence of up to 15 characters that may be assigned to one of the ten function keys. When an assigned key is pressed, the string is input to GWBASIC.

Assigning a null string (0 length) disables the function key as a soft key.

The defined string expression is input as a GWBASIC command when the assigned function key is pressed.

If the value returned for *key number* is not in the range 1 to 10, an illegal function call error occurs. The previous KEY string assignment is retained.

#### KEY ON

This is the initial setting. Causes the KEY values to be displayed on the 25th line.

10 KEY values are displayed if screen width is 80 and 5 KEY values are displayed if 40. In each case, only the first 6 characters are displayed.

#### KEY OFF

Erases the KEY display from the 25th line. The function key assignment text is not eliminated when the display is erased. It may be used as is.

#### KEY LIST

List all 10 KEY values on the screen. All 15 characters of each value are displayed.

The default definitions of Function Keys are as follows. **RETURN** indicates that a Carriage Return is entered.

<b>F1</b>	LIST	<b>F6</b>	, "LPT1:" RETURN
<b>F2</b>	RUN RETURN	<b>F7</b>	TRON RETURN
<b>F3</b>	LOAD"	<b>F8</b>	TROFF RETURN
<b>F4</b>	SAVE"	<b>F9</b>	KEY
<b>F5</b>	CONT RETURN	<b>F10</b>	SCREEN 0,0,0 RETURN

When a soft key is assigned, the **INKEY\$** function returns one character of the Soft Key string per invocation.

#### **Example**

```
100 KEY 1, "FILES"  
110 KEY ON
```

## KEY [*n*] ON/OFF/STOP

### Purpose

Enables, disables, or terminates interrupts caused by a specified key.

### Syntax

```
KEY (n) ON
KEY (n) OFF
KEY (n) STOP
```

### Comments

*n* indicates the number of the function key that causes the interrupt.

Assigns enabling (ON), disabling (OFF), and termination (STOP) of interrupts that occur when the key indicated by *n* is pressed. The keys indicated by (*n*) are listed below.

1-10	Function Keys <b>F1-F10</b>
11	Cursor Up Arrow
12	Cursor Left Arrow
13	Cursor Right Arrow
14	Cursor Down Arrow
15-20	Keys defined by the form: KEY ( <i>n</i> ) CHR\$(SHIFT)+CHR\$( <i>scancode</i> )

The KEY (*n*) ON statement enables interrupts. After this command is executed, an interrupt is generated each time an assigned Key is pressed. If the first line number (other than 0) of an interrupt handling routine is assigned in an ON KEY...GOSUB statement, a check is performed to see if the assigned key is pressed each time GWBASIC executes a statement. If the key is pressed, GWBASIC branches to the subroutine with the assigned line number.

The KEY (*n*) OFF statement disables interrupt. If the key is pressed, it is ignored.

The KEY (*n*) STOP statement prevents interrupts from occurring when the key is pressed. However, they are recorded in memory and occur if KEY (*n*) is activated later.

KEY (*n*) ON has no effect on displaying the assignment text of the function key on the bottom line of the screen.

### **Example**

```
100 KEY (1) ON
110 ON KEY (1) GOSUB 160
120 'LOOP
130 A$=CHR$(RND*100)
140 PRINT A$;
150 GOTO 130
160 'WAIT
170 FOR I=1 TO 1000
180 NEXT
190 PRINT "      "
200 FOR I=1 TO 1000
210 NEXT
220 RETURN
```

### **Reference**

ON KEY  
Appendix J

## **KILL**

### **Purpose**

Deletes a file from disk

### **Syntax**

KILL *pathname*  
KILL *filename*

### **Comments**

If a KILL statement is given for a file that is currently OPEN, a File already open error occurs.

KILL is used for all types of disk files: program files, random data files, and sequential data files.

### **Example**

```
200 KILL "DATA1"
```

### **Reference**

Appendix A

## **LCOPY**

### **Purpose**

Prints text currently displayed on the screen.

### **Syntax**

LCOPY

### **Comments**

The LCOPY statement sends the active page of text screen to the current printer device.

## LET

### Purpose

Assigns the value of an expression to a variable.

### Syntax

[LET] *variable* = *expression*

### Comments

Notice the word LET is optional; the equal sign also assigns an expression to a variable name.

### Examples

```
110 LET D = 12
120 LET E = 12^2
130 LET F = 12^4
140 LET SUM = D+E+F
```

.  
.  
.

or

```
110 D = 12
120 E = 12^2
130 F = 12^4
140 SUM = D+E+F
```

## LINE

### Purpose

Draws straight lines and rectangles on the screen in graphics mode.

### Syntax

```
LINE [(x1,y1)]-(x2,y2) [, [color ][, B[F]][, style ]]
```

### Comments

(x1,y1) and (x2,y2) are shown on the screen as absolute or relative coordinates. For *color* display, refer to Using the Screen, Graphics Mode in Chapter 1.

With the LINE statement, it is possible to draw straight lines connecting the coordinates designated by (x1,y1) and (x2,y2).

The simplest form of line is:

```
LINE -(x2,y2)
```

This draws from the last point to the point (x2,y2) in the foreground attribute.

For example, if LINE -(100,100) is executed after LINE (0,0)-(10,10), a line can be drawn connecting (0,0)-(10,10)-(100,100).

The coordinates for the final point can be written in relative form. In this case, the final point becomes the point defined by adding a specified offset to the coordinates of the first point. For example, the result of LINE (100,100)-STEP(20,-20) is the line segment (100,100)-(120,80).

The permissible range of the coordinate depends on graphics mode. Refer to the SCREEN statement for details.

The final argument to LINE is **B** (box) or **BF** (filled box). You can leave out the attribute argument and include the final argument as follows:

```
LINE (0,0)-(100,100),,B      'Draw box in foreground or include it
```

```
LINE (0,0)-(200,200),2,BF    'Draw filled box attribute 2
```



The **,B** tells GWBASIC to draw a rectangle with the points (x1,y1) and (x2,y2) as opposite corners. This avoids giving the four LINE commands that follow to perform the same function:

```
LINE (x1,y1)-(x2,y1)
LINE (x1,y1)-(x1,y2)
LINE (x2,y1)-(x2,y2)
LINE (x1,y2)-(x2,y2)
```

The **,BF** tells GWBASIC to draw a rectangle as in **,B** but also fill in the interior with the selected attribute.

*Style* is a 16-bit integer mask used when putting down pixels on the screen. This is called line-Styling.

Each time LINE stores a point on the screen, it uses the current circulating bit in *style*. If that bit is 0, no store is done. If the bit is 1, then a normal store occurs. After each point, the next bit position in *style* is selected.

Since a 0 bit in *style* does not clear out the old contents, you may wish to draw a background line before a styled line in order to force a known background.

*Style* is used for normal lines and boxes, but has no effect on filled boxes.

### Examples

```
100 SCREEN 1,0: COLOR 0,0
110 FOR I = 1 TO 10
120 LINE (0,0)-(RND*319,RND*199),RND*2+1
130 NEXT
140 FOR I = 1 TO 10
150 LINE (319,199)-(RND*319,RND*199),RND*2+1
160 NEXT
170 GOTO 110
```

Draws colored lines from two corners of your screen.

```
LINE (0,0) - (160,100),3,,& HFF00
```

Draws a dashed line from the upper left corner to the screen center.

### Reference

COLOR, SCREEN

## LINE INPUT

### Purpose

Inputs an entire line (up to 254 characters) to a string variable, without the use of delimiters.

### Syntax

```
LINE INPUT[;]["prompt string ";] string variable
```

### Comments

The *prompt string* is a string displayed at the terminal before input is accepted. A question mark is not displayed unless it is part of the *prompt string*. All input from the end of the prompt to the **RETURN** is assigned to *string variable*. If a line feed/carriage return sequence (this order only) is encountered, both characters are echoed, but the carriage return is ignored, the line feed is put into *string variable*, and data input continues.

If **LINE INPUT** is immediately followed by a semicolon, then pressing **RETURN** to end the input line does not echo a carriage return/line feed sequence at the terminal.

A **LINE INPUT** command can be stopped by pressing **CTRL BREAK**. Then **GW BASIC** returns to the command level and displays the **Ok** prompt. Typing **CONT** resumes execution at the **LINE INPUT**.

### Example

See example under **LINE INPUT#**.

## LINE INPUT#

### Purpose

Reads an entire line (up to 254 characters), without delimiters, from a sequential disk data file into a string variable.

### Syntax

LINE INPUT# *file number, string variable*

### Comments

*File number* is the number under which the file was OPENed. *String variable* is the variable name that the line is assigned. LINE INPUT# reads all characters in the sequential file up to a return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a GWBASIC program saved in ASCII mode is being read as data by another program.

### Example

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1,C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT#1,C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES 234,4 MEMPHIS
LINDA JONES 234.4 MEMPHIS
Ok
```

## LIST

### Purpose

Allows a program to be listed to the screen or another device.

### Syntax

```
LIST [line number [-line number ]][, dev ]
```

### Comments

*Line number* is a valid line number in the range 0 to 65529. *Dev* is a valid device specification.

If the optional device specification is omitted, the specified lines are displayed.

Listings directed to the screen by omitting the device specification may be paused at any time by pressing **CTRL NUM LOCK**. Listings directed to devices can not be interrupted. That is, *LIST range* can be interrupted, but not *LIST range*, "SCRN:".

Pressing **CTRL BREAK** terminates a screen listing.

If the line range is omitted, the entire program is listed.

If only *line number* is specified, only that line is displayed.

When the hyphen (-) is used in a line range, three options are available:

1. If only the first number is given, that line and all higher numbered lines are listed.
2. If only the second number is given, all lines from the beginning of the program through the given line are listed.
3. If both numbers are specified, the inclusive range is listed.

If LIST is followed by a space and a period (.), the line at the present cursor position is displayed.

When files are listed to the disk, they are saved in ASCII format. Files saved in this manner can later be MERGED.

## **Examples**

LIST

Lists the program currently in memory.

LIST 500

Lists line 500.

LIST 150-

Lists all lines from 150 on the end of the program.

LIST -1000

Lists all lines from the lowest number through 1000.

LIST 150-1000

Lists lines 150 through 1000.

## **LLIST**

### **Purpose**

Prints all or part of the program currently in memory on the printer.

### **Syntax**

LLIST [*line number* [-[ *line number* ]]]

### **Comments**

LLIST assumes a 132-character wide printer.

GWBASIC always returns to command level after LLIST is completed. The options for LLIST are the same as for LIST.

### **Examples**

See the examples for LIST.

## LOAD

### Purpose

Loads a file from disk into memory.

### Syntax

LOAD *filespec* [, R ]

### Comments

*Filespec* is the name of the disk file. MS-DOS adds a default extension (.BAS) to the file. The LOAD command:

- o Closes all open files.
- o Deletes all variables.
- o Deletes the current program from memory.
- o Loads the designated program.

If you use the R option with LOAD, the program is RUN after it is LOAded, and all open data files are kept open. Thus, LOAD with the R option may be used to chain several programs (or segments of the same program). Information is passed between the programs using their disk data files.

### Example

```
LOAD "STRTRK",R
```

## LOCATE

### Purpose

Moves the cursor to a spot on the active screen. Optional parameters set the cursor to blink on and off and define its size.

### Syntax

```
LOCATE [row ][,col ][,cursor ][,start ][,stop ]]]
```

### Comments

*Row* is where you want to place the cursor. This is a numeric expression in the range 1 to 25.

*Col* is where you want to place the cursor. This is a numeric expression in the range 1 to 10 or 1 to 80 depending upon your video display type and screen width.

*Cursor* determines whether the cursor is visible or not. You can enter either:

0	No display
1	Display

*Start* is the cursor starting scan line, a numeric expression in the range 0 to 15.

*Stop* is the cursor stop scan line, a numeric expression in the range 0 to 15.

Any parameter may be omitted. Omitted parameters assume the last value.

The desired size of the cursor can be obtained by using *start* and *stop*.

If you give the *start* scan line parameter and omit the *stop* scan line parameter, *stop* assumes the *start* value. Using a color monitor, the bottom scan line is 7. Also, if *start* is given as a higher value than *stop*, the cursor disappears.

The cursor is not displayed during ordinary program execution. In order to display it, enter **LOCATE,,1**.

Any values entered outside of these ranges result in an illegal function call error. Previous values are retained.



## **LPRINT and LPRINT USING**

### **Purpose**

Prints data at the line printer.

### **Syntax**

```
LPRINT [list of expressions]  
LPRINT USING string exp; list of expressions
```

### **Comments**

This command functions similar to PRINT and PRINT USING, except that output goes to the line printer. See PRINT and PRINT USING for more information.

LPRINT assumes an 80-column printer.

## **LSET and RSET**

### **Purpose**

Moves data from memory to a random file buffer (in preparation for a PUT statement).

### **Syntax**

LSET *string variable* = *string expression*  
RSET *string variable* = *string expression*

### **Comments**

If *string expression* requires fewer bytes than were FIELDed to *string variable*, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before using LSET or RSET. See the MKI\$, MKS\$, MKD\$ functions in Chapter 3 for more information.

**Note:** LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N\$ in a 20-character field. This is useful for formatting printed output.

### **Examples**

```
150 LSET A$=MKS$<(AMT)
160 LSET D$=DESC($)
```

### **Reference**

Appendix B

## **MERGE**

### **Purpose**

Merges a specified disk file into the program currently in memory.

### **Syntax**

MERGE *filename*

### **Comments**

*Filename* is the name used when the file was SAVED. (With MS-DOS, the extension .BAS is supplied.) The file must have been SAVED in ASCII format. (If not, a Bad file mode error occurs.)

If any lines in the disk file have the same line numbers as lines in the program memory, the lines from the file on disk replace the corresponding lines in memory (merging may be thought of as "inserting" the program lines on disk into the program in memory).

GWBASIC always returns to command level after executing a MERGE command.

### **Example**

MERGE "NUMBR5"

## MID\$

### Purpose

Replaces a portion of one string with another string.

### Syntax

MID\$( *string exp1* ), *n* [, *m* ]) = *string exp2*

### Comments

The characters in *string exp1*, beginning at position *n*, are replaced by the characters in *string exp2*.

The optional *m* refers to the number of characters from *string exp2* that are used in the replacement. If *m* is omitted, all of *string exp2* is used.

However, regardless of whether *m* is omitted or included, the replacement of characters never goes beyond the original length of *string exp1*.

### Example

```
10 A$="KANSAS CITY, MO"  
20 MID$(A$,14)="KS"  
30 PRINT A$  
RUN  
KANSAS CITY, KS
```

### Reference

The MID\$ function also returns a substring of a given string. See Chapter 3 for more information.

## **MKDIR**

### **Purpose**

Makes a new directory.

### **Syntax**

MKDIR *pathname*

### **Comments**

*Pathname* is a string expression up to 128 characters long that identifies the new directory to be created. MKDIR works exactly like the MS-DOS command MKDIR.

### **Examples**

MKDIR "SALES"

Creates subdirectory SALES in the root directory of the current drive.

MKDIR "ACCOUNTING"

Creates subdirectory ACCOUNTING in the root directory of the current drive.

MKDIR "B:INVENTORY"

Creates subdirectory INVENTORY in the root directory on Drive B.

## NAME

### Purpose

Changes the name of a disk file.

### Syntax

NAME *old filename* AS *new filename*

### Comments

*Old filename* must exist and *new filename* must not exist; otherwise an error results. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Pathnames are not allowed. That is, only files in a current directory may be renamed. Specifying a path results in a Bad filename error.

Rules to keep in mind when using NAME are:

1. *Oldname* and *newname* must be closed before rename. The same open file check is used as in OPEN and KILL.
2. There must be one free file handle for performing the open check.

### Example

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

This example shows that the file formerly named ACCTS is now named LEDGER.

## **NEW**

### **Purpose**

Deletes the program currently in memory and clears all variables.

### **Syntax**

NEW

### **Comments**

NEW is entered at command level to clear memory before entering a new program. GWBASIC always returns to command level after a NEW command is entered.

## ON COM

### Purpose

Defines the starting line of the subroutine used when data arrives at the communications buffer.

### Syntax

ON COM(*port number*) GOSUB *line number*

### Comments

*Port number* is the number of the communications adapter being used. *Line number* is the starting line of communications handling routine. COM ON must be executed before executing ON COM.

ON COM only assigns the starting line of the subroutine, therefore execution of this command does not by itself result in branching to the subroutine. Branching to the subroutine after an interrupt only occurs when an outside factor (in this case, the arrival of data at the communications buffer) is present.

Execution of COM STOP disables interrupts from the communications buffer, but the arrival of data is recorded. Therefore, if data arrives after execution of COM STOP and then COM ON is executed subsequently, GWBASIC goes immediately to the *line number* assigned by ON COM. You can disable communications buffer interrupts by assigning line 0 as the branching destination.

COM STOP is executed automatically whenever an interrupt occurs. This is to prevent recalling of the interrupt handling subroutine if another interrupt occurs during execution of the subroutine (the branching destination assigned by ON COM). Unless COM OFF is executed during the handling subroutine, COM ON is executed automatically with the execution of RETURN at the end of the handling subroutine, re-enabling interrupts from the communications buffer. Interrupts only occur while GWBASIC is executing a program. If an ERROR interrupt occurs, all other interrupts (ERROR, PEN, COM, KEY) are disabled.

To specifically assign the destination for return from the handling subroutine, use RETURN *line number*. If possible, handle all of the data at the communications buffer at the same time with the handling subroutine. If characters are handled one at a time and the baud rate of the communications circuit is high, the result could be a large increase in overhead time and possibly a buffer overflow.

### Reference

COM



## ON ERROR GOTO

### Purpose

Enables error trapping and specifies the first line of the error handling subroutine.

### Syntax

ON ERROR GOTO *line number*

### Comments

Once error trapping has been enabled, all errors that are detected, including direct mode errors (syntax errors), cause a jump to the specified error handling subroutine. If line number does not exist, an Undefined line error results.

To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes GWBASIC to stop and print the error message for the error that caused the trap.

It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered that has no recovery action.

**Note:** If an error occurs during execution of an error handling subroutine, the GWBASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

### Example

```
10 ON ERROR GOTO 1000
```

## ON...GOSUB and ON...GOTO

### Purpose

Branches to one of several specified line numbers, depending on the value returned when an expression is evaluated.

### Syntax

```
ON expression GOTO list of line numbers  
ON expression GOSUB list of line numbers
```

### Comments

The value of *expression* determines the *line number* in the list used for branching. For example, if the value is 3, the third *line number* in the list is the destination of the branch (if the value is a non-integer, the fractional portion is rounded).

In the ON...GOSUB statement, each *line number* in the list must be the first line number of a subroutine. *Line numbers* must be separated by commas.

If the value of *expression* is zero or greater than the number of items in the list (but less than or equal to 255), GWBASIC continues with the next executable statement. If the value of *expression* is negative or greater than 255, an illegal function call error occurs.

### Example

```
100 ON L-1 GOTO 150,300,320,390
```

## ON KEY

### Purpose

Defines the starting line of the subroutine used when a KEY interrupt occurs.

### Syntax

ON KEY *key number* GOSUB *line number*

### Comments

*Key number* is the function key number causing the interrupt/branch to the specified subroutine.

Key interrupts can be disabled by assigning line 0 as the branch destination. KEY *key number* ON must be used before an ON KEY command.

ON KEY only defines the starting line of the subroutine, so execution of this command does not by itself result in branching to the subroutine. Branching to the subroutine after an interrupt only occurs when an outside factor (in this case, pressing a key) is present.

Execution of KEY *key number* OFF disables interrupts from the key designated by *key number*.

Execution of KEY *key number* STOP disables interrupts from the key designated by *key number*. Pressing the key is recorded, so if you press the key after execution of KEY *key number* STOP and then subsequently execute KEY *key number* ON, GWBASIC goes immediately to the line number defined by ON KEY.

Also, KEY *key number* STOP is executed automatically whenever interrupts occur. This prevents recalling of the interrupt handling subroutine (the destination assigned by ON KEY) if the same Key is depressed again during execution of the subroutine. Unless KEY *key number* OFF is executed during the handling subroutine, KEY *key number* ON is executed automatically with the execution of RETURN at the end of the subroutine.

Interrupts only occur while GWBASIC is executing a program. Also, if an ERROR interrupt occurs, all other interrupts (ERROR, PEN, COM, KEY) are disabled.

INPUT\$ and INKEY\$ cannot be used to identify the Key which caused the interrupt. Therefore, ON KEY must be used to define a handling subroutine for each Key separately.

To specifically assign the line number for return from the handling subroutine, use RETURN *line number*.

The ON KEY(*n*) statement allows 6 additional user defined KEY traps. This allows any key or **CTRL** and **SHIFT** key combinations to be trapped as follows:

ON KEY(*i*) GOSUB *line number*

*i* is an integer expression dependent on the number of function keys and direction keys. GWBASIC has ten function keys and four direction keys. Therefore, you can define keys 15 through 20.

User defined keys are defined by the statement:

KEY *i*, CHR\$(*j*)+CHR\$(*k*)

*i* is the user key number (15 through 20).

*j* and *k* are intended to uniquely define a key on the keyboard. GWBASIC interprets as follows.

*j* is the mask for the latched keys: **CAPS LOCK**, **NUM LCK**, **ALT**, **CTRL**, Left **SHIFT**, and Right **SHIFT**. The appropriate bit information must be set in order to trap a key that is Shifted, Ctrl-Shifted, or Alt-Shifted. Keyboard Flag values in HEX are:

<b>CAPS LOCK</b>	&H40	<b>CAPS LOCK</b> is active
<b>NUM LCK</b>	&H20	<b>NUM LCK</b> is active
<b>ALT</b>	&H08	The <b>ALT</b> key is depressed
<b>CTRL</b>	&H04	The <b>CTRL</b> key is depressed
Left <b>SHIFT</b>	&H02	The left hand <b>SHIFT</b> key is depressed
Right <b>SHIFT</b>	&H01	The right hand <b>SHIFT</b> key is depressed

*k* is the number identifying one of the 83 physical keys on the keyboard.

These rules apply to keys trapped by GWBASIC:

1. The **PRTSC** key, the Line Printer echo toggle is pressed first. Defining the **PRTSC** key as a user defined Key Trap doesn't prevent characters from being echoed to the Line Printer if depressed.
2. Function keys and cursor keys are examined next. Defining a function key or cursor key as a Key Trap has no effect because they are considered pre-defined.
3. User defined keys are examined next.
4. Any key that is trapped is not passed on to GWBASIC.

**Note:** The user-defined key traps apply to any combination of keys that you define, including **CTRL BREAK** and **CTRL ALT DEL** (system reset). They may be used as an aid to prevent you from accidentally stopping a program or rebooting the computer.

### **Example**

```
100 FOR I=1 TO 3
110 KEY (I) ON
120 NEXT I
130 ON KEY (1) GOSUB 170
140 ON KEY (2) GOSUB 180
150 ON KEY (3) GOSUB 190
160 GOTO 100
170 PRINT "GOTO";:RETURN
180 PRINT "PRINT";:RETURN
190 PRINT "AUTO";:RETURN
```

Press **CTRL** and **BREAK** together to exit the program.

### **Reference**

KEY, RETURN  
Appendix J

## ON PEN

### Purpose

Transfers control to a specified line number when the light pen is activated.

### Syntax

ON PEN GOSUB *line number*

### Comments

A PEN ON statement must be executed before using ON PEN. *Line number* is the first line number of the light pen event trap routine.

If you specify a *line number* of 0, light pen trapping is disabled. If you specify a valid *line number* with ON PEN, GWBASIC checks to see if the light pen was activated before processing each new statement.

If the light pen was activated, GWBASIC transfers program control to the subroutine starting at the specified *line number*. A PEN STOP statement is automatically executed, to prevent a loop from occurring. You should use a RETURN statement to return to the line following the ON PEN statement; use RETURN *line number* to return to another program location. When GWBASIC returns from the trap routine, a PEN ON statement is automatically executed (unless you place a PEN OFF statement in the trap routine).

If PEN OFF has been executed prior to the ON PEN statement, GWBASIC does not execute a GOSUB when the light pen is activated. The event is not stored in memory.

If PEN STOP has been executed prior to the ON PEN statement, no GOSUB is executed, but the event is stored in memory so that trapping will take place when PEN ON is next executed.

### Example

```
50  PEN ON
60  ON PEN GOSUB 200
.
.
200 REM Pen trap subroutine
.
.
250 RETURN
```

Transfers control to the subroutine at line 200 when the light pen is activated. GWBASIC returns to line 60 upon completion.

## ON PLAY

### Purpose

Causes an event trap when the Music Background queue goes from  $n$  notes to  $n-1$  notes.

### Syntax

ON PLAY ( $n$ ) GOSUB *line number*

### Comments

$n$  is an integer expression in the range 1 to 32. Values outside this range result in an illegal function call error.

*Line number* is the statement line number of the Play event trap subroutine.

PLAY ON	Enables PLAY event trapping.
PLAY OFF	Disables PLAY event trapping.
PLAY STOP	Suspends PLAY event trapping.

Rules to remember when using PLAY:

1. A PLAY event trap is issued only when playing in Music Background mode (i.e., PLAY "MB."). Play event traps are not issued when running in Music Foreground mode (i.e., default case, or PLAY "MF..").
2. A PLAY event trap is not issued if the Music Background queue is already empty when a PLAY ON is executed.
3. Choose conservative values for  $n$ . An *ON PLAY* (32).. statement causes event traps so often that there is little time left for program execution.

## ON STRIG

### Purpose

Transfers control to a specified line number when one of the joystick triggers is activated.

### Syntax

```
ON STRIG(n) GOSUB line number
```

### Comments

A STRIG(*n*) ON statement must be executed before using ON STRIG(*n*). Replace the *n* parameter with the number of the joystick trigger to be trapped:

0	Indicates trigger A1
2	Indicates trigger B1
4	Indicates trigger A2
6	Indicates trigger B2

*Line number* is the first line number of the joystick event trap routine. If you specify a *line number* of 0, joystick trapping is disabled. If you specify a valid *line number* with ON STRIG(*n*), GWBASIC checks to see if the joystick trigger specified in *n* was activated before processing each new statement.

If the joystick trigger was activated, GWBASIC transfers program control to the subroutine starting at the specified *line number*. A STRIG(*n*) STOP statement is automatically executed, to prevent a loop from occurring. You should use a RETURN statement to return to the line following the ON STRIG(*n*) statement; use RETURN *line number* to return to another program location. When GWBASIC returns from the trap routine, a STRIG(*n*) ON statement is automatically executed (unless you place a STRIG(*n*) OFF statement in the trap routine).

If STRIG(*n*) OFF has been executed prior to the ON STRIG(*n*) statement, GWBASIC does not execute a GOSUB when the joystick trigger is activated. The event is not stored in memory.

If STRIG(*n*) STOP has been executed prior to the ON STRIG(*n*) statement, no GOSUB is executed, but the event is stored in memory so that trapping will take place when STRIG(*n*) ON is next executed.

Event trapping is performed only during program execution. If an ON ERROR statement produces an error trap, all event trapping for ON PEN, ON PLAY, ON COM, ON TIMER, and other ON PEN and ON STRIG statements is disabled.



### **Example**

```
50  STRIG(2) ON
60  ON STRIG(2) GOSUB 200
.
.
200 REM Joystick trigger B1 trap subroutine
.
.
250 RETURN
```

Transfers control to the subroutine at line 200 when joystick trigger B1 is activated. Upon completion of the trap routine, GWBASIC returns to line 60.

### **Reference**

STRIG(n)

## ON TIMER

### Purpose

Causes an event trap every *n* seconds.

### Syntax

ON TIMER(*n*) GOSUB *line number*

### Comments

*n* is a numeric expression in the range 1 through 864000 (1 second through 24 hours). Values outside this range result in an illegal function call error.

*Line number* is the statement line number of the TIMER event trap subroutine.

TIMER ON	Enables TIMER event trapping.
TIMER OFF	Disables TIMER event trapping.
TIMER STOP	Suspends TIMER event trapping.

### Example

```
10 ON TIMER(60) GOSUB 10000
20 TIMER ON
.
.
.
10000 OLDROW = CSRLIN           'Save current row
10010 OLDCOL = POS(0)          'Save current column
10020 LOCATE 1,1:PRINT TIMES;
10030 LOCATE OLDROW,OLDCOL      'Restore Row and Column
10040 RETURN
```

Displays the time of day on Line 1 every minute

## OPEN

### Purpose

Opens a file.

### Syntax

OPEN *file mode 1* ,[#] *file number* ,*filespec* [, *reclen* ]

OPEN *filename* FOR *file mode 2* AS [#] *file number* [ LEN = *reclen* ]

### Comments

In the first format, *file mode 1* is a string expression whose first character is one of the following:

O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random input/output mode.

In the second format, *file mode 2* is a string constant -- INPUT, OUTPUT, or APPEND -- specified without double quotation marks ( " " ):

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
APPEND	Specifies sequential output where the file is positioned to the end of data on the file when it is opened.

If *file mode 2* is left out, random access becomes the default mode.

*File number* is an integer expression returning a number in the range 1 through 255.

*Filespec* specifies the name of the file to be opened.

*Reclen* is an integer expression in the range 1 through 32767. This value sets the record length to be used for random files (see the FIELD statement). If omitted, the record length defaults to 128 byte records.

The OPEN statement allocates a buffer for input/output to the file and determines the mode of access that is used with the buffer. Once a file has been opened, *file number* can be used by I/O. The OPEN statement must be used before executing the following commands:

PRINT #	WRITE #
PRINT # USING	INPUT #
INPUT #	GET
LINE INPUT #	PUT

GET and PUT commands can be used only in files opened in random access mode and only disk files can be opened in this manner.

When no device name is given, the active drive is assumed.

A File not found error is displayed if the input file does not exist. If the output file does not exist, one is created.

See OPEN COM for more details on communication files.

Only one file per *file number* can be OPENed.

OPEN allows *pathname* in place of *filespec*.

OPEN permits a file to be opened for INPUT, OUTPUT, or APPEND regardless of whether a LEN= option is present.

**Note:** You may not OPEN a file for OUTPUT or APPEND if the file is already open in any mode. Since it is possible to reference the same file in a subdirectory via different paths, it is nearly impossible for GWBASIC to know that it is the same file simply by looking at the path. For this reason, GWBASIC does not let you open the file for OUTPUT or APPEND if it is on the same disk, even if the path is different. For example, if SALES is your current directory, the following commands all refer to the same file.

```
OPEN "O", #1, "REPORT"
OPEN "R", #3, "\DIVISION\SALES\REPORT"
OPEN "I", #4, "\SALES\REPORT"
OPEN "O", #2, "..\..\SALES\REPORT"
```

## Reference

OPEN COM

## OPEN COM

### Purpose

Opens a communication file.

### Syntax

```
OPEN "DEV :[ speed [, parity [, data [, stop ]]]]  
[, LF ][, RS ][, CS [ n ]][, DS [ n ]] [ CD [ n ][ PE ] "  
AS [#] file number [ LEN = number ]
```

### Comments

OPEN COM opens files for RS-232C communications adapters.

*DEV*: is a valid communications device. Valid devices are:

COM1: COM2: COM3:

*Speed* is a literal integer specifying the transmit/receive baud rate.  
Valid speeds are: 75, 110, 150, 300, 600, 1200, 2400, 4800, 9600.

*Parity* is a one-character constant specifying the parity for transmit and receive as follows:

O	ODD	Odd transmit/receive parity checking
E	EVEN	Even transmit/receive parity checking
N	NONE	No transmit parity, no receive parity checking.
S	SPACE	Parity bit is always transmitted and received as space (0 bit).
M	MARK	Parity bit is always transmitted and received as mark (1 bit).

*Data* is an integer constant indicating the number of transmit/receive data bits. Valid values are: 4, 5, 6, 7, or 8.

*Stop* is an integer constant indicating the number of stop bits. Valid values are: 1 or 2. If omitted, then 110 bps transmit two stop bits, all others transmit one stop bit.

The LF, RS, CS, DS, CD, and PE options affect the line signals. These options may appear in any order in the OPEN COM statement. These are:

#### LF

Used when communications file data is being printed on a serial line printer. A line feed character (0A hex) is automatically appended to a carriage return character (0C hex).

The LF option used in the OPEN statement also affects how data is read with the INPUT# and LINE INPUT# statements. Reading stops when a carriage return character is encountered.

The line feed character is ignored by the INPUT# and LINE INPUT# statements.

## RS

Suppresses the request-to-send control character (1E hex). If the RS option is not used, the RTS action occurs automatically when the communications file is opened.

## CS[n]

Controls clear-to-send. [n] specifies the number of milliseconds that pass before the host times out. Valid [n] values are 0 to 65535, inclusive. The default value is 1000. If no [n] value is specified or the value equals 0, the line status is not checked.

Subsequent communications files I/O statements won't work if the CS[n] option is not selected.

## DS[n]

Controls the Data set ready message. [n] specifies the number of milliseconds that pass before the host times out. Valid [n] values range from 0 to 65535, inclusive. The default value is 1000. If no [n] value is specified or the value equals 0, the line status is not checked.

Subsequent communications file I/O statements won't work if the DS[n] option is not selected.

## CD[n]

Controls the Carrier detect message. [n] specifies the number of milliseconds that pass before the host times out. Valid [n] values range from 0 to 65535, inclusive. The default value is 0. If no value is given or the default is accepted, the line status is not checked.

The CD option is often referred to as the RLSD (Received line signal detect) message.

## PE

Activates parity checking. If the PE option is omitted, the default (no parity check) is active. If a parity error is detected, a Device I/O error message appears. The high order bit is turned for 7 or fewer data bits. Framing and overrun errors always turn on the high order bit and generate a Device I/O error message.

*Number* is the maximum number of bytes that can be read from the communications buffer when using the GET or PUT statement. The default is 128 bytes.

When CTRL Z is received, it is treated as the end of the file. If a file is closed, CTRL Z is received as a command. TAB changes position and carriage return is treated as a new line. When LF is assigned, it is paired with each carriage return.

*File number* is an integer expression returning a valid file number. The number is then associated with the file for as long as it is OPEN and is used to refer other COM I/O statements to the file.

Missing parameters invoke the following defaults:

Speed = 300bps  
Parity = EVEN  
Bits = 7

OPEN COM is very similar to the OPEN command except that it applies to input and output of the RS-232C communications adapter.

If you are transmitting or receiving numeric data, you must specify 8 data bits.

A COM device may be OPENed to only one file number at a time.

A Device timeout error occurs if the RS-232C asynchronous communications adapter is not installed correctly.

## Reference

OPEN

## **OPTION BASE**

### **Purpose**

Declares the minimum value for array subscripts.

### **Syntax**

OPTION BASE *n*

### **Comments**

The default value of *n* is 0. If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is 1.

The CHAINED program may now have an OPTION BASE statement if no arrays are passed. The CHAINED program inherits the OPTION BASE value of the CHAINing program.

**Note:** In GWBASIC 1.0, OPTION BASE gave an error even if the new value was the same as the old.



## **OUT**

### **Purpose**

Sends a byte to a machine output port.

### **Syntax**

OUT *I*,*J*

### **Comments**

*I* and *J* are integer expressions in the range 0 to 65535. *I* is the port number; *J* is the data to be transmitted.

OUT is the complementary statement to the INPUT statements.

### **Example**

100 OUT 12345,225

In assembly language, this is equivalent to:

```
MOV DX,12345
MOV AL,225
OUT DX,AL
```

## PAINT

### Purpose

Fills an arbitrary graphics figure of the specified fill attribute.

### Syntax

```
PAINT (xstart,ystart)[,paint attribute  
[,border attribute [,background attribute ]]]
```

### Comments

*Xstart* and *ystart* are the coordinates of the starting point within the area to be filled. They can be noted in either absolute or relative form.

The *paint attribute* chooses from the attribute range for the current screen mode. The *border attribute* sets the border color of the figure to be filled in the attribute range for the current screen mode. You must specify both paint color and border color. *Background* is a 1-byte string expression used in paint tiling. See page 2-95. Refer also to the COLOR statement for detail.

PAINT must start on a non-border point, otherwise PAINT has no effect.

PAINT supports "tiling" -- the repetition of a pattern along a row and down the columns of the screen. Like LINE, PAINT looks at a tiling mask each time a point is put down on the screen.

If *paint attribute* is omitted, the standard foreground attribute is used. If *paint attribute* is a numeric formula, then the number must be a valid color and is used to paint the area as before.

PAINT can fill any figure, but PAINTing jagged edges or very complex figures may result in an Out of Memory error. If this happens, you must use the CLEAR statement to increase the amount of stack space available.

The *paint attribute* defaults to the foreground attribute if not given, and the *border attribute* defaults to the *paint attribute*.

If *paint attribute* is a string formula, then tiling is performed as follows:

The tile mask is always 8 bits wide and may be from 1 to 64 bytes long. Each byte in the tile string masks 8 bits along the x axis when putting down points. Each byte of the tile string is rotated as required to align along the y axis such that the tile byte mask = y MOD tile length.

This is done so that the tile pattern is replicated uniformly over the entire screen (as if a PAINT 0,0) were used.

x,y	x increases bit of tile byte								
	8	7	6	5	4	3	2	1	
0,0	x	x	x	x	x	x	x	x	Tile byte 1
0,1	x	x	x	x	x	x	x	x	Tile byte 2
0,2	x	x	x	x	x	x	x	x	Tile byte 3
.									
.									
0,63	x	x	x	x	x	x	x	x	Tile byte 64

**Table 2-1** PAINT Tiling Patterns

In high-resolution mode (SCREEN 2), the screen can be painted with x's using the following statement:

```
PAINT (320,100), CHR$(&H81) + CHR$(&H42) + CHR$(&H24) + CHR$(&H18) +
CHR$(&H18) + CHR$(&H24) + CHR$(&H42) + CHR$(&H81)
```

This appears on the screen as:

0,0	x					x	CHR\$(&H81)
							Tile byte 1
0,1		x				x	CHR\$(&H42)
							Tile byte 2
0,2			x			x	CHR\$(&H24)
							Tile byte 3
0,3				x	x		CHR\$(&H18)
							Tile byte 4
0,4				x	x		CHR\$(&H18)
							Tile byte 5
0,5			x			x	CHR\$(&H24)
							Tile byte 6
0,6		x				x	CHR\$(&H42)
							Tile byte 7
0,7	x					x	CHR\$(&H81)
							Tile byte 8

Since there are 2 bits per pixel in medium-resolution (SCREEN 1), each byte of the tile pattern only describes 4 pixels. In this case, every 2 bits of tile byte describe 1 of the 4 possible colors associated with each of the 4 pixels to be put down.

*Background attribute* is a string formula returning one character. When omitted, the default is CHR\$(0).

When supplied, *background attribute* specifies the background tile pattern or color byte to skip when checking for boundary termination.

Occasionally, you may want to tile paint over an already painted area that is the same color as two consecutive lines in the tile pattern. Normally, PAINT quits when it encounters two consecutive lines of the same color as the point being set (the point is surrounded). It is not possible to draw alternating blue and red lines on a red background without this parameter. PAINT stops as soon as the first red pixel is drawn. Specifying red [CHR\$(&HAA)] as the background attribute allows the red line to be drawn over the red background.

You cannot specify more than two consecutive bytes that match the background attribute in the tile string. Specifying more than two bytes results in an illegal function call error.

For additional information about tiling, see the file TILE.TXT on the GWBASIC 3.11 disk.

## **PALETTE**

This statement is not supported in GWBASIC 3.11. Colors are specified through graphics statements or the COLOR statement.

## PALETTE USING

This statement is not supported in GWBASIC 3.11. Colors are specified through graphics statements or the COLOR statement.

## **PEN**

### **Purpose**

Reads the light pen.

### **Syntax**

PEN ON  
PEN OFF  
PEN STOP  
 $x = \text{PEN}(n)$

### **Comments**

$x$  is the numeric variable receiving the PEN value

$n$  is a valid numeric expression returning an unsigned integer in the range 0 to 9

$x = \text{PEN}(n)$  reads the light pen coordinates, where  $n$  is:

- 0 Returns -1 if pen was down since last poll, 0 if not.
- 1 Returns the x pixel coordinate when pen was last pressed. Range 0 to 319 (medium resolution), or 0 to 639 (high resolution).
- 2 Returns the y pixel coordinate when pen was last pressed. Range 0 to 199 (medium resolution).
- 3 Returns the current pen switch value: -1 if down, 0 if up.
- 4 Returns the last known valid x pixel coordinate. Range 0 to 319 (medium resolution) or 0 to 639 (high resolution).
- 5 Returns the last known valid y pixel coordinate. Range 0 to 199 (medium resolution).
- 6 Returns the character row position when pen was last pressed. Range 1 to 24.
- 7 Returns the character column position when pen was last pressed. Range 1 to 40 or 1 to 80, depending on SCREEN.
- 8 Returns the last known valid character row. Range 1 to 24.
- 9 Returns the last known valid character column position. Range 1 to 40 or 1 to 80, depending on SCREEN.

The PEN function is initially off. A pen read function call results in an illegal function call error. A PEN ON statement must be executed before any pen read function calls can be made.

Conversely, for execution speed improvements, it is a good idea to turn the pen off with a PEN OFF statement for those programs not using the Light Pen.

#### **Example**

```
100 CLS:LOCATE,,0
110 PEN ON
120 ON PEN GOSUB 140
130 GOTO 120
140 X=PEN(1):Y=PEN(2)
150 PSET (X,Y)
160 RETURN
```

#### **Reference**

ON PEN



## PLAY

### Purpose

To play melody indicated by character string.

### Syntax

PLAY *string expression*

### Comments

The single character commands in PLAY are:

A-G[ #, +, or - ]

Play the note indicated by the letter. A number sign (#) or plus sign (+) following the letter means sharp, and a minus sign (-) means flat.

*Ln*

Length. Sets the length of each note. L4 is a quarter note, L1 is a whole note, etc. *n* may range from 1 to 64.

The length may also follow the note when it is desired to change the length only for the note. In this case, A16 is equivalent to L16A.

MF

Music Foreground. Music (PLAY statement) and SOUND are to run in Foreground. That is, each subsequent note or sound does not start until the previous note or sound is finished.

It is possible to escape from the PLAY statement by pressing **CTRL BREAK**.

MB

Music Background. Music (PLAY statement) and SOUND are to run in background. That is, each note or sound is placed in a buffer allowing the GWBASIC program to continue execution while music plays in the background. Up to 32 notes (or rests) can be played in background at a time.

MN

Music Normal. Each note plays 7/8ths of the time determined by L (length).

ML

Music Legato. Each note plays the full period set by L (length).

MS

Music Staccato. Each note plays 3/4ths of the time determined by L (length).

Nn

Play note  $n$ .  $n$  may range from 0 to 84. In the 7 possible octaves, there are 84 notes.  $n=0$  means rest.

On

Octave. Sets the current octave. There are 7 octaves (0-6). Each octave begins with C and ends with B.

Pn

Pause. P may range from 1 to 64. Length is the same as Ln. Please refer to the section.

Tn

Tempo. Sets the number of L4's in a minute.  $n$  may range from 32 to 255. Default is 120.

Dot or Period. After each note, causes the note to play  $3/2$  times the period determined by, L (length) times T (tempo). Multiple dots may appear after the note. The period is scaled accordingly (For example: A. =  $3/2$ , A.. =  $9/4$ , A... =  $27/8$ , etc.). Dots may appear after a pause (P) and scale the pause length as described above.

< >

Decrement (<) and increment (>) octave. Decrement octave does not decrement below octave 0, and increment does not increment above octave 6.

Xstring

Execute substring.

The  $n$  in each command can be given either as a constant or a variable. Also, the X command can be used to change the tempo or the octave in a repeat PLAY.

## Examples

```
10 PLAY "MBMNT12OL16"  
20 FOR I=1 TO 5  
30 PLAY "O=I;CDEFGAB"  
40 NEXT
```

```
PLAY "<<"    'Decrement Octave twice  
PLAY ">A"    'Increment Octave and play note A
```

## **POKE**

### **Purpose**

Writes a byte into a memory location.

### **Syntax**

POKE *I*,*J*

### **Comments**

The integer expression *I* is the address of the memory location to be POKEd. The integer expression *J* is the data to be POKEd. *I* must be in the range 0 to 65536; *J* must be in the range 0 to 255.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See PEEK in Chapter 3 for more information.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

### **Example**

```
10 POKE &H5A00,&HFF
```

## PRINT

### Purpose

Outputs data at a terminal.

### Syntax

PRINT [ *list of expressions* ]

### Comments

If *list of expressions* is omitted, a blank line is printed. If *list of expressions* is included, the values of the expressions is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions (strings must be enclosed in quotation marks).

**Print Positions.** The position of each printed item is determined by the punctuation used to separate the items in the list. GWBASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, GWBASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8(-7) is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .00000000000000001 and 1D-16 is output as 1D-16. A question mark may be used in place of the word PRINT in a PRINT statement.

## Examples

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
10          0          -25          3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
RUN
? 9
  9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

Press the CTRL and BREAK keys together to exit from this program.

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
RUN
 5   10   10   20   15   30   20   40   25   50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. A number is always preceded by a space and positive numbers are preceded by a space. In line 40, a question mark is used instead of the word PRINT.

## PRINT USING

### Purpose

Prints strings or numbers using a specified format.

### Syntax

PRINT USING *string exp; list of expressions*

### Comments

*List of expressions* comprises the string expressions or numeric expressions that are to be printed, separated by semicolons. *String exp* is a string constant (or variable) comprising special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

**String Fields.** When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

!

Specifies that only the first character in the given string is to be printed.

\n spaces\

Specifies that 2+n characters from the string are to be printed. If the backslash characters are typed with no spaces, two characters are printed; with one space, three characters are printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

&

Specifies a variable length string field. When the field is specified with &, all of the characters you assigned to the string are printed.

### Examples

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\  \";A$;B$;"!!"
RUN
LO
LOOK OUT
LOOK OUT  !!
```

```

10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT

```

**Numeric Fields.** When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

#

A number sign (#) is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit is always printed (as 0 if necessary). Numbers are rounded as necessary. For example:

```

PRINT USING "##.##";.78
0.78

```

```

PRINT USING "###.##";987.654
987.65

```

```

PRINT USING "##.##  ";10.2,5.3,66.789,.234
10.20  5.30  66.79  0.23

```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+

A plus sign (+) at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.

-

A minus sign (-) at the end of the format field causes negative numbers to be printed with a trailing minus sign. For example:

```

PRINT USING "+##.##  ";-68.95,2.4,55.6,-.9
-68.95  +2.40  +55.60  -0.90

```

```

PRINT USING "##.##-  ";-68.95,22.449,-7.01
68.95-  22.45  7.01-

```



**\*\***

A double asterisk (\*\*) at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The \*\* also specifies positions for two more digits. For example:

```
PRINT USING "***.# ";12.39,-0.9,765.1
*12.4  *-0.9  7651.
```

**\$\$**

A double dollar sign (\$\$) causes a dollar sign to be printed to the immediate left of the formatted number. The \$\$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with \$\$\$. Negative numbers cannot be used unless the minus sign trails to the right. For example:

```
PRINT USING "$$###.##";456.78
$456.78
```

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format. For example:

```
PRINT USING "####,.##";1234.5
1,234.50
```

**^^^^**

Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position is used to the left of the decimal point to print a space or a minus sign. For example:

```
PRINT USING "##.##^^^^";234.56
2.35E+02
```

```
PRINT USING ".####^^^^-";888888
.8889E+06
```

```
PRINT USING "+.##^^^^";123
+.12E+03
```

—

An underscore ( \_ ) in the format string causes the next character to be output as a constant character.

```
PRINT USING " |##.##!";12.34  
|12.34|
```

The constant character itself may be an underscore by placing two underscores ( \_\_ ) in the format string.

%

If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign is printed in front of the rounded number.

```
PRINT USING "##.##";111.22  
%111.22
```

```
PRINT USING ".##";.999  
%1.00
```

If the number of digits specified exceeds 24, an illegal function call error results.

## PRINT# and PRINT# USING

### Purpose

Writes data to a sequential disk file.

### Syntax

PRINT#*file number*, [ USING *string expression*; ] *list of expressions*

### Comments

*File number* is the number used when the file was OPENed for output. *String expression* comprises formatting characters as described in PRINT USING. The expressions in a *list of expressions* are the numeric and/or *string expressions* that are written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it is input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

If commas are used as delimiters, the extra blanks that are inserted between print fields are also written to disk.

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, if A\$="CAMERA" and B\$="93604-1" then the statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;" ";B$
```

The image written to disk now is CAMERA,93604-1, which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, using CHR\$(34).

For example, if A\$="CAMERA, AUTOMATIC" and B\$="93604-1", the statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes "CAMERA, AUTOMATIC" 93604-1" to disk.

The succeeding statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A\$ and " 93604-1" to B\$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$$###.##,";J;K;L
```

## Reference

See WRITE# in this chapter for more information. For more examples using PRINT#, see Appendix A.

## PSET and PRESET

### Syntax

```
PSET (x,y)[,color]  
PRESET (x,y)[,color]
```

### Purpose

Draws a dot at the assigned position on the screen.

### Comments

(x,y) are coordinates for drawing (setting) a dot; may be either absolute or relative.

*Color* assigns dot color. The range of color value depends on graphics mode. Refer to the CIRCLE statement in detail.

PRESET has an identical syntax to PSET. The only difference is that if no third parameter is given, the background color 0 (zero) is selected. When a third argument is given, PRESET is identical to PSET.

If an out of range coordinate is given to PSET or PRESET, no action is taken nor is an error given. If *color* is greater than the permitted value, this results in an illegal function call error.

### Example

```
100 SCREEN 2  
110 FOR X=0 TO 100  
120 Y=(X*X)/100  
130 PSET (X,Y)  
140 NEXT
```

## PUT (Files)

### Purpose

Writes a record from a random buffer to a random disk file.

### Syntax

```
PUT [#] file number [, record number ]
```

### Comments

*File number* is the number under which the file was OPENed. If *record number* is omitted, the record has the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

**Note:** PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement. In the case of WRITE#, GWBASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a Field overflow error.

### Examples

See Appendix A.

## **PUT (COM)**

### **Purpose**

Allows fixed length I/O for COM.

### **Syntax**

PUT *file number*,*nbytes*

### **Comments**

*File number* is an integer expression returning a valid file number.

*nbytes* is an integer expression returning the number of bytes to be transferred into or out of the file buffer.

## PUT (Graphics)

### Purpose

Outputs graphic patterns in the assigned position on the screen.

### Syntax

PUT(*x,y*), *array name* [, *operation* ]

### Comments

(*x,y*) are coordinates of the upper left-hand corner of the rectangular region on the screen.

*Array name* is the name of the numerical array containing the graphic pattern being output on the screen.

In contrast to GET, PUT causes the array data to be output on the screen.

*Operation* is the assigned operation to be performed with data already displayed on the screen when a graphic pattern is output on the screen. Operations include PSET, PRESET, XOR (the default if not specified), OR, and AND:

#### PSET

Outputs the graphic pattern contained in the array on the screen as is (the opposite operation from GET).

#### PRESET

Reverses the graphic pattern contained in the array and outputs it on the screen (similar to a photographic negative).

#### AND

Outputs to the screen the result of combining the graphic pattern contained in the array and the data already displayed on the screen, on a 1 to 1 basis.

#### OR

The graphic pattern in the array is output on the screen overlapping the data already displayed there.



## XOR

The result of performing the XOR operation on the data on the screen and the graphic pattern in the array is output on the screen. In other words, if points are already displayed (set) at certain coordinates, the data that corresponds to the graphic pattern contained in the array are reversed (NOT) at those points.

If points are not set, the data corresponding to the array are displayed directly on the screen. Assigning the XOR function is effective in displaying patterns moving over a background.

If the output graphic pattern is larger than the screen size, it results in an illegal function call error.

## Reference

GET

## RANDOMIZE

### Purpose

Reseeds the random number generator.

### Syntax

RANDOMIZE [*expression*]

### Comments

If *expression* is omitted, GWBASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

RANDOMIZE with no arguments prompts you for a new seed. RANDOMIZE *expression* no longer forces floating point values to integer values. *Expression* may be any numeric formula.

To get a new random seed without prompting, you can use the new numeric TIMER function as *expression*, as in the second example below.

### Example

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3 (user types 3)
.88598 .484668 .586328 .119426 .709225
Ok
RUN
Random Number Seed (-32768 to 32767)? 4 (user types 4 for new sequence)
.803506 .162462 .929364 .292443 .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3 (same sequence as first RUN)
.88598 .484668 .586328 .119426 .709225
Ok
```

## READ

### Purpose

Reads values from a DATA statement and assigns them to variables.

### Syntax

READ *list of variables*

### Comments

A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables must be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a Syntax error results. (See DATA for more information.)

A single READ statement may access one or more DATA statements (they are accessed in order), or several READ statements may access the same DATA statement. If the number of variables in list of variables exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

### Examples

```
.  
.   
.   
80 FOR I=1 TO 10  
90 READ A(I)  
100 NEXT I  
110 DATA 3.08,5.19,3.12,3.98,4.24  
120 DATA 5.08,5.55,4.00,3.16,3.37  
.   
.   
. 
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, and so on.

```
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,",COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY          STATE          ZIP
DENVER,       COLORADO       80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

## REM

### Purpose

Allows explanatory remarks to be inserted in a program.

### Syntax

REM *remark*

### Comments

REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution continues with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark ( ' ) instead of :REM.

---

### WARNING

Do not use REM in a data statement because it will be considered legal data.

---

### Examples

```
.  
. 120 REM CALCULATE AVERAGE VELOCITY  
130 FOR I=1 TO 20  
140 SUM=SUM + V(I)  
.  
.
```

or,

```
.  
. 120 FOR I=1 TO 20 'CALCULATE AVERAGE VELOCITY  
130 SUM=SUM + V(I)  
140 NEXT I  
.  
.
```

## RENUM

### Purpose

Renumbers program lines.

### Syntax

RENUM [[*new number* ][, [*old number* ][, *increment* ]]]

### Comments

*New number* is the first line number to be used in the new sequence. The default is 10. *Old number* is the line in the current program where renumbering is to begin. The default is the first line of the program. *Increment* is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message Undefined line xxxxx in yyyy is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyy may be changed.

**Note:** RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10,20,and 30) or to create line numbers greater than 65529. An illegal function call error results.

### Examples

RENUM

Renumbers the entire program. The first new line number is 10. Lines increment by 10.

RENUM 300,,50

Renumbers the entire program. The first new line number is 300. Lines increment by 50.

RENUM 1000,900,20

Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

## **RESET**

### **Purpose**

Closes all disk files and deletes all system buffers.

### **Syntax**

RESET

### **Comments**

If all open files are on disks, the RESET command is the same as the CLOSE command with no file numbers specified.

### **Reference**

CLOSE

## RESTORE

### Purpose

Allows DATA statements to be reread from a specified line.

### Syntax

RESTORE [*line number*]

### Comments

After you execute a RESTORE statement, the next READ statement accesses the first item in the program's first DATA statement. If *line number* is specified, the next READ statement accesses the first item in the specified DATA statement.

The CHAIN performs a RESTORE function before running the CHAINED program. This resets the pointer to the DATA statements.

### Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
.
.
.
```



## RESUME

### Purpose

Continues program execution after an error recovery procedure has been performed.

### Syntax

```
RESUME  
RESUME 0  
RESUME NEXT  
RESUME line number
```

### Comments

Any one of the four formats shown above may be used, depending upon where execution is to resume:

```
RESUME
```

or

```
RESUME 0
```

Execution resumes at the statement which caused the error.

```
RESUME NEXT
```

Execution resumes at the statement immediately following the one which caused the error.

```
RESUME line number
```

Execution resumes at *line number*.

A RESUME statement that is not in an error trap routine causes a RESUME without error message to be printed.

### Example

```
10  ON ERROR GOTO 900  
.  
.  
900 IF (ERR=230)AND(ERL=90) THEN PRINT  
    "TRY AGAIN":RESUME 80  
.  
.
```

## RETURN

### Purpose

Returns program control after completion of a subroutine.

### Syntax

RETURN [*line number*]

### Comments

The RETURN statement is used following a subroutine to return program control to the line number following the GOSUB statement that called the subroutine.

RETURN *line number* allows non-local returns from any subroutine. Replace *line number* with the number of the line to which you want to return.

This form of RETURN was especially designed for use with event-trapping routines. As the last statement in an event-trapping routine, RETURN *line number* lets you return to the GWBASIC program at a point that eliminates the GOSUB entry that the trap created.

You should use the non-local form of RETURN carefully, because all other GOSUB, WHILE, and FOR statements active at the time of the trap remain active.

### Example

```
10 OPEN "EMPDATA" FOR INPUT AS 1
.
.
.
100 GOSUB 1200 'Get employee data
110 WRITE #1, NAME$, ADDR$, HIREDATE$
.
.
.
1200 INPUT "EMPLOYEE NAME";NAME$
1210 INPUT "ADDRESS";ADDR$
1220 INPUT "HIREDATE";HIREDATE$
1230 RETURN
```

### Reference

GOSUB...RETURN

## **RMDIR**

### **Purpose**

Removes a directory.

### **Syntax**

RMDIR *pathname*

### **Comments**

*Pathname* is a string expression not exceeding 128 characters that identifies the subdirectory to be removed from its parent. RMDIR works exactly like the MS-DOS command RMDIR.

### **Example**

RMDIR "SALES"

Removes a subdirectory called SALES from a directory called WEST/REPORT/SALES (assuming your current directory is the root directory).

## RUN

### Purpose

Executes the program currently in memory. With the R option, loads a file from disk into memory and executes it.

### Syntax

```
RUN  
RUN [line number]  
RUN filename [, R]
```

### Comments

If *line number* is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. GWBASIC always returns to command level after a RUN is executed.

If *filename* is specified, it is the name used when the file was SAVEd. (with MS-DOS, the default extension .BAS is supplied).

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the R option, all data files remain OPEN.

### Examples

```
RUN
```

```
RUN "NEWFIL",R
```

### Reference

Appendix A

## SAVE

### Purpose

Saves a program file on disk.

### Syntax

```
SAVE filename  
SAVE filename [, A ]  
SAVE filename [, P ]
```

### Comments

*Filename* is a quoted string conforming to MS-DOS's filenames requirements (with MS-DOS, the default extension .BAS is added). If *filename* already exists, the file is written over.

The program is saved to the file specified in *filename*. If the device name is omitted the active drive in use is assumed.

If *filename* is less than 1 character or greater than 8 characters in length, a Bad file name error is issued and the save is aborted.

Use the A option to save the file in ASCII format. Otherwise, GWBASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. You can RUN or LOAD a file protected with the P option normally, but any attempt to view the program using LIST or to change it using EDIT causes an illegal function call error to be issued. You cannot unprotect a file saved with the P option.

### Examples

```
SAVE "MYFILE"  
  
SAVE "COM2",A  
  
SAVE "PROG",P
```

### Reference

Appendix B

## SCREEN

### Purpose

Sets screen attributes.

### Syntax

```
SCREEN [mode ][,burst ][,apage ][,vpage ]]
```

### Comments

*Mode* specifies the screen mode:

- 0 Elementary text (40 or 80 characters)
- 1 Elementary graphics 320x200
- 2 Elementary graphics 640x200
- 3 Extended graphics 640x200 (sixteen color)  
Extended graphics 720x348 (two color)

1, 2, and 3 above are valid when a color display is connected. *Mode* = 3 operates in two color mode when the hardware for the active screen is set to MONO and in sixteen color mode when the hardware for the active screen is set to COLOR.

*Burst* is a numeric expression that is either true or false. This enables or disables the color mode. False (zero) in the text mode (*mode* = 0) disables color (only black and white), and true (non-zero) enables it. In medium resolution (*mode* = 1), true (non-zero) disables color, and false (zero) enables it. In high resolution (*mode* = 2), only black and white are available, so this parameter is ineffective.

*Apage* (active page) selects the page that is written by the output statement for the screen. This parameter is specified by a numeric expression containing a value 0 to -1 (for the maximum page, see Chapter 1.)

*Vpage* (visual page) indicates the page to be displayed on the screen and is specified in the same manner as *apage*. *Vpage* may be different from *apage*.

If the specification of *vpage* is omitted, it is assumed to be the same as *apage*. If all parameters are valid, the new screen mode is stored in memory and the current screen is erased. Foreground color is set to white while background color and border color are set to black. If the new screen mode is the same as the old screen mode, nothing changes. If only *apage* and *vpage* are specified in the text mode, the displayed page is changed. At the time GWBASIC is started, both pages are zeros. Using *apage* and *vpage*, a page can be displayed on the screen while another is being created.

**Note:** Each parameter can be omitted. When it is omitted, the old value remains except for *vpage*. If a value outside the allowable range is specified, an illegal function call error is caused. In such a case, the old value remains.

#### **Examples**

10 SCREEN 0, 1, 0, 0

The color text mode is selected. *Apage* and *vpage* are both zero.

20 SCREEN ,, 1, 3

*Mode* and *burst* remain unchanged. *Apage* is 1 and *vpage* is 3.

## SHELL

### Purpose

Loads and executes another program (.COM, .EXE, or .BAT) file. When the program finishes, control returns to the GWBASIC program at the statement following the SHELL statement. A program executed under control of GWBASIC is called a *child process*.

### Syntax

SHELL [*command string*]

### Comments

Child processes (or children) are executed when the SHELL statement loads and runs a copy of COMMAND.COM. This is the same as the MS-DOS COMMAND /C option to load a secondary command processor. By using COMMAND in this way, you may execute internal MS-DOS commands like DIR, PATH, COPY, and SORT and redirect standard input and output. Any program parameters are automatically routed to the default file control blocks (FCBs).

**Note:** SHELL searches for COMMAND.COM on the drive and directory specified in the COMSPEC statement in your environment. Before using the SHELL statement, make sure there is a copy of the MS-DOS file COMMAND.COM in the root directory of the drive MS-DOS was booted from. For example, if you booted MS-DOS from Drive A, there should be a copy of COMMAND.COM on your BASIC disk.

If you enter SHELL with no *command string*, a copy of COMMAND.COM is loaded and the MS-DOS prompt appears. You can enter any internal MS-DOS commands. To return to GWBASIC, type EXIT at the command line and press RETURN. *Command string* contains the name of the program you want to run. Optionally, the string may contain parameters to the child program.

If no extension is specified for the program, COMMAND.COM looks for a .COM file, then an .EXE file, and then a .BAT file. If no matching files are found, SHELL issues a File not found error message.

Arguments separated from the program name by at least one blank space are processed as replaceable parameters to the program specified in *command string*.

When a SHELL statement is executed, GWBASIC remains in memory while the child process is running. When the child process finishes, GWBASIC continues.

If you are using SHELL to execute a .BAT file, it must contain the EXIT command as the last line of the batch file.



You cannot use `SHELL` to load another copy of `GWBasic`. If you attempt to run `GWBasic` as a child process, the You cannot `SHELL` to `BASIC` message is displayed.

Because child processes may open files and use devices differently than the parent program, `GWBasic` cannot totally protect its environment during `SHELL` operations. The following guidelines will help to prevent programs running under `SHELL` from harming the `GWBasic` environment.

1. If possible, preserve the current state of all hardware during a `SHELL` command. It is recommended that you refrain from using certain devices within child processes.
  - o Child processes may modify screen mode parameters. To insure that you return to `GWBasic` in the screen mode you expect, follow the `SHELL` statement in your program with `CLS`.
  - o Interrupt vectors used by `GWBasic` are saved when you execute `SHELL`. However, you should save and restore any interrupt vectors the child must use. The child process may perform this task.
  - o Certain devices (interrupt controller, counter timer, DMA controller, I/O latch, and `USART`) are placed in a specific state by `GWBasic`, and should be left untouched by the child process.
2. If a child process alters a file opened by the parent `GWBasic` program, the application can fail or produce unpredictable results. If it is necessary to update such files, they should be closed in the parent process **before** the `SHELL` statement is executed. Reopen the files upon returning to `GWBasic`.
3. Before executing a `SHELL` statement, `GWBasic` frees any memory that is not being used. If you start `GWBasic` with the `/M` option, however, `GWBasic` assumes that you are attempting to load an assembly language routine outside of its memory block. This prevents `GWBasic` from "compressing the workspace" before executing `SHELL` and may result in an Out of memory error.

The preferred method is to load assembly language routines **before** starting `GWBasic`. This is done by placing `INT 27H` at the end of your assembly language routine. The `INT 27H` code allows the routine to terminate as stay resident. For example:

CSEG	SEGMENT CODE
	;Machine language subroutine
	RET ;Last instruction
START:	
	INT 27h ;Terminate, stay resident
CSEG	ENDS
	END START

Be sure to load these subroutines by running them before you start GWBASIC. You can do this by placing them in an AUTOEXEC.BAT file.

A child process should never terminate and stay resident. Doing so may not leave enough memory for GWBASIC to expand its workspace to the original size. If GWBASIC cannot restore the workspace, it closes all files, displays the error message Can't continue after SHELL, and exits to MS-DOS.

#### **Example**

```
100 OPEN "MAILLIST.DAT" FOR OUTPUT AS #4
.
.
.
200 CLOSE 4
210 SHELL "SORT <MAILLIST.DAT >MAILSORT.DAT"
220 OPEN "MAILSORT.DAT" FOR INPUT AS #5
```

Writes program data from lines 40-100 to file MAILLIST.DAT for sorting into file MAILSORT.DAT.

## SOUND

### Purpose

Generates sound through the speaker.

### Syntax

SOUND *freq,duration*

### Comments

*Freq* is the desired frequency in Hertz (cycles per second). This must be an unsigned integer in the range 37 to 32767.

*Duration* is the desired duration in Clock ticks. *Duration* must be a numeric expression in the range 0 to 65536.

Current clock ticks occur 18.2 times per second.

There is no break in program execution when sound is generated with the SOUND statement. Execution is continued until the next SOUND statement.

If *duration* is 0, any current SOUND statement that is running is turned off. If no SOUND statement is running, SOUND *x*, 0 has no effect.

When *duration* is not 0, the next SOUND statement is not executed until the sound generated by the previous SOUND statement ends.

In order to obtain an interval in which no sound is produced, assign 32767 as *freq*.

### Example

```
100 SOUND RND*1500,INT(RND*10)
110 GOTO 100
```

## **STOP**

### **Purpose**

Terminates program execution and returns to command level.

### **Syntax**

STOP

### **Comments**

STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close files.

GWBASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

### **Example**

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
30.76923
Ok
CONT
115.9
Ok
```

## **STRIG**

### **Purpose**

Enables and disables event trapping when the joystick triggers are pressed.

### **Syntax**

```
STRIG(n) ON  
STRIG(n) OFF  
STRIG(n) STOP
```

### **Purpose**

You must execute a STRIG(*n*) ON statement to enable event trapping with the ON STRIG(*n*) statement. Replace the *n* parameter with the number of the joystick trigger to be trapped:

- 0 Indicates trigger A1
- 2 Indicates trigger B1
- 4 Indicates trigger A2
- 6 Indicates trigger B2

After a STRIG(*n*) ON statement, GWBASIC checks to see if the joystick trigger specified in *n* was activated before processing each new statement.

If STRIG(*n*) OFF is executed, GWBASIC does not execute a GOSUB when the joystick trigger is activated. The event is not stored in memory.

If STRIG(*n*) STOP is executed, no trapping takes place, but the event is stored in memory so that trapping will take place when STRIG(*n*) ON is next executed.

## SWAP

### Purpose

Exchanges the values of two variables.

### Syntax

SWAP *variable, variable*

### Comments

Any type *variable* may be SWAPed (integer, single precision, double precision, string), but the two *variables* must be of the same type or a Type mismatch error results.

### Example

```
LIST
10 A$=" ONE " : B$=" ALL " : C$=" FOR "
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
ONE FOR ALL
ALL FOR ONE
Ok
```

## **SYSTEM**

### **Purpose**

Escapes from GWBASIC and returns to MS-DOS.

### **Syntax**

SYSTEM

### **Comments**

When SYSTEM is executed, all files are closed before returning to MS-DOS.  
**All programs and data in memory are lost, so use care.**

## **TIME\$**

### **Purpose**

Sets the time.

### **Syntax**

TIME\$=*string expression*

### **Comments**

TIME\$ complements the TIME\$ function, which retrieves the time. *String expression* returns one of the following:

hh                Sets the hour; minutes and seconds default to 0.

hh:mm            Sets the hour and minutes; seconds default to 0.

hh:mm:ss        Sets the hour, minutes, and seconds.

A 24-hour time notation is used. 8:30 p.m. is entered as 20:30:00.

### **Example**

```
TIME$="20:30:00"
```

The current time is set to 8:30 p.m.



## **TIMER**

### **Purpose**

TIMER ON enables real time event trapping.  
TIMER OFF disables real time event trapping.  
TIMER STOP suspends real time event trapping.

### **Syntax**

TIMER ON  
TIMER OFF  
TIMER STOP

### **Comments**

The TIMER ON statement enables real time event trapping by an ON TIMER statement. GWBASIC checks between every statement to see if the timer has reached the specified level. If it has, the ON TIMER statement is executed.

TIMER OFF disables the event trap. If an event takes place, it is not remembered if a subsequent TIMER ON is used.

TIMER STOP disables the event trap, but if an event occurs, it is remembered and an ON TIMER statement is executed as soon as trapping is re-enabled.

### **Reference**

ON TIMER

## TRON and TROFF

### Purpose

Traces the execution of program statements (TRaceON/TRaceOFF).

### Syntax

```
TRON
TROFF
```

### Comments

Used as an aid in debugging, the TRON statement is executed in either the direct or indirect mode. It enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

### Example

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

## VIEW

### Purpose

Defines a *physical viewport* limit from Vx1,Vy1 (upper left x,y coordinates) to Vx2,Vy2 (lower right x,y coordinates).

### Syntax

```
VIEW [[screen][Vx1,Vy1)-(Vx2,Vy2)[,[fill]][,[border]]]]
```

### Comments

The x and y coordinates must be within the physical bounds of the screen and define the rectangle within the screen that graphics will map into. Initially, RUN, or VIEW with no arguments defines the entire screen as the viewport.

*Fill* allows you to fill the view area with a color. If *fill* is omitted, the view area is not filled.

*Border* lets you draw a line surrounding the viewport if space for a border is available. If *border* is omitted, no border is drawn.

For VIEW (Vx1,Vy1)-(Vx2,Vy2) , all points plotted are relative to the viewport. That is, Vx1 and Vy1 are added to the x and y coordinates before putting down the point on the screen. If the statement:

```
VIEW(10,10)-(200,100)
```

is executed, the point set down by the statement PSET (0,0),3 is at the physical screen location 10,10. For the statement:

```
VIEW SCREEN (Vx1,Vy1)-(Vx2,Vy2)
```

all coordinates are absolute and may be inside or outside of the screen limits, but only those within the VIEW limits are plotted. If:

```
VIEW SCREEN (10,10)-(200,100)
```

is executed, the point set down by the statement PSET (0,0),3 does not appear because 0,0 is outside of the viewport. PSET (0,0),3 is within the viewport, and places the point in the upper-left hand corner.

## WAIT

### Purpose

Suspends program execution while monitoring the status of a machine input port.

### Syntax

WAIT *port number*, *I* [, *J*]

### Comments

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern.

*I* and *J* are integer expressions. The data read at the port is exclusively OR'ed with the integer expression *J*, and then AND'ed with *I*. If the result is zero, GWBASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *J* is omitted, it is assumed to be zero.

---

### Caution

It is possible to enter an infinite loop with the WAIT statement. If an infinite loop is encountered during program execution, you can proceed in one of two ways: terminate the program execution by pressing the CTRL and BREAK keys, or restart your computer manually by pressing CTRL, ALT, and DEL, or the RESET button.

---

### Example

```
100 WAIT 32,2
```

## WHILE...WEND

### Purpose

Executes a series of statements in a loop as long as a given condition is true.

### Syntax

```
WHILE expression
.
.
[loop statements]
.
.
WEND
```

### Comments

If *expression* is not zero (i.e., true), *loop statements* are executed until the WEND statement is encountered. GWBASIC then returns to the WHILE statement and checks *expression*. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE...WEND loops may be nested to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a WHILE without WEND error, and an unmatched WEND statement causes a WEND without WHILE error.

### Example

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115     FLIPS=0
120     FOR I=1 TO J-1
130         IF A$(I)>A$(I+1) THEN
            SWAP A$(I),A$(I+1):FLIPS=1
140     NEXT I
150 WEND
```

## WIDTH

### Purpose

Sets the printed width in number of characters for the screen and line printer.

### Syntax

```
WIDTH size
WIDTH file number, size
WIDTH dev, size
```

### Comments

*Size* must be a numeric expression in the range 0 through 255. This is a new width.

*File number* must be a numeric expression (integer) in the range 1 through 255.

*Dev* must be a valid string expression for the device identifier. Valid devices are: SCRN:, LPT1:, LPT2:, LPT3:, COM1:, COM2:, and COM3:.

GW BASIC automatically adds a carriage return and line feed when the specified width in the WIDTH statement is reached.

Depending upon the device specified, the following actions are possible:

```
WIDTH size
```

or

```
WIDTH "SCRN:", size
```

This sets the screen width. Only 40- or 80-column width is allowed.

**Note:** Changing the screen width causes the screen to be cleared.

If the screen is in Medium Resolution Graphics Mode (SCREEN 1), WIDTH 80 forces the screen into High Resolution Graphics Mode (SCREEN 2).

If the screen is in High Resolution Graphics Mode (SCREEN 2), WIDTH 40 forces the screen into Medium Resolution Graphics Mode (SCREEN 1).

WIDTH *dev, size*

This form of WIDTH stores the new width value without actually changing the current width setting. A subsequent OPEN, LIST "LPTn:" (n=1-3), or LLIST statement uses this value and set the new width to it.

WIDTH *file number, size*

The *file number* resets the currently open file to the output width.

WIDTH cannot be executed from the keyboard (KYBD:).

A file may be associated with LPT1:, LPT2:, COM1:, COM2:, and COM3:.

No error message appears, even if the specified print width is larger than can be accepted by the printer in use.

Specifying WIDTH 255 for the Line Printer disables line folding. This has the effect of infinite width.

## Reference

SCREEN

## WINDOW

### Purpose

Draws lines, graphs, or objects in space not bounded by the physical limits of the screen.

### Syntax

```
WINDOW [[SCREEN] (Wx1,Wy1)-(Wx2,Wy2)]
```

### Comments

The WINDOW statement defines the window transformation from Wx1,Wy1 (upper left x,y coordinates) to Wx2,Wy2 (lower right x,y coordinates). This is done by using arbitrary coordinates you select called *world coordinates*.

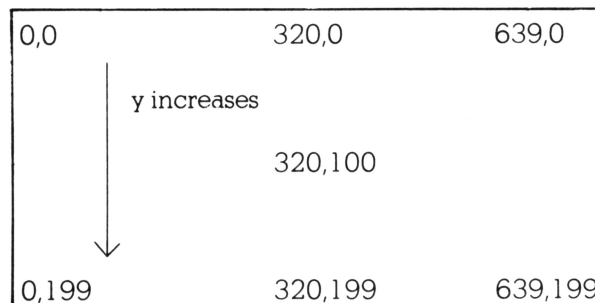
A world coordinate is any valid single precision floating point number pair. GWBASIC then converts world coordinate pairs into the appropriate physical coordinate pairs for subsequent display with screen space. To make this transformation from world space to the physical space of the viewing screen, you must know which portion of the unbounded (floating point) world coordinate space contains the information you want to be displayed. This rectangular region is called a *window*.

The x and y coordinates may be any single precision floating point number. They define the world coordinate space that graphics will map into the physical coordinate space defined by the VIEW statement.

WINDOW inverts the y coordinate on subsequent graphics statements. This allows the screen to be viewed in true cartesian coordinates. If you execute the command:

```
NEW  
SCREEN 2
```

the screen appears as:

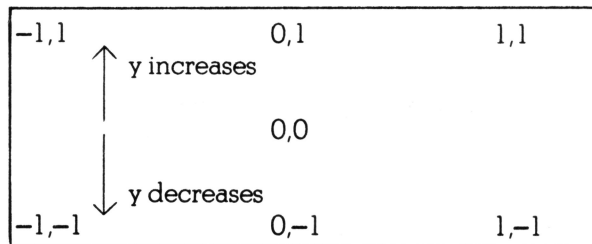




If you then execute the command:

```
WINDOW (-1,-1)-(1,1)
```

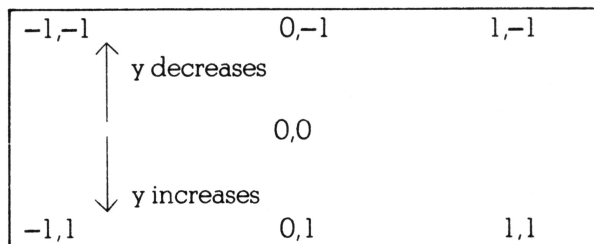
the screen appears as:



The WINDOW SCREEN variant does not invert the y coordinate. If you execute the command:

```
WINDOW SCREEN (-1,-1)-(1,1)
```

the screen appears as:



WINDOW with no arguments (or RUN) disables window transformation.

## WRITE

### Purpose

Outputs data at the terminal.

### Syntax

WRITE [*list of expressions*]

### Comments

If *list of expressions* is omitted, a blank line is output. If *list of expressions* is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item is separated from the last by a comma.

Printed strings are delimited by quotation marks. After the last item in the list is printed, GWBASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement.

### Example

```
10 A=80:B=90:C$="THAT'S ALL"  
20 WRITE A,B,C$  
RUN  
80,90,"THAT'S ALL"  
Ok
```

## **WRITE#**

### **Purpose**

Writes data to a sequential file.

### **Syntax**

*WRITE#file number,list of expressions*

### **Comments**

*File number* is the number under which the file was OPENed in O mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for you to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

### **Examples**

If A\$="CAMERA" and B\$="93694-1", the statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would put "CAMERA" to A\$ and "93604-1" to B\$.

### CHAPTER 3

#### GW BASIC FUNCTIONS AND VARIABLES

The intrinsic functions provided by GW BASIC are presented in this chapter. The way these functions are presented is the same as in Chapter 2. The GW BASIC functions, unlike user-defined functions, may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

$X$ and $Y$	Represent any numeric expressions
$I$ and $J$	Represent integer expressions
$X\$$ and $Y\$$	Represent string expressions

If a floating point value is supplied where an integer is required, GW BASIC rounds the fractional portion and uses the resulting integer.

**Note:** With the GW BASIC interpreter, only integer and single precision results are returned by functions.

As an exception to the description of the /D command line option in Chapter 1, some transcendental functions with a double precision argument are evaluated in double precision.

## **ABS**

### **Syntax**

ABS ( X )

### **Comments**

Returns the absolute value of the expression X.

### **Example**

```
PRINT ABS(7*(-5))
35
Ok
```

## **ASC**

### **Syntax**

ASC ( X\$ )

### **Comments**

Returns a numerical value that is the ASCII code of the first character of the string X\$.

If X\$ is null, an illegal function call error is returned. See Appendix I for ASCII codes.

### **Example**

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
84
Ok
```

### **Reference**

See the CHR\$ function for ASCII-to-string conversion.  
See Appendix I for a list of ASCII codes.

## ATN

### Syntax

ATN ( X )

### Comments

Returns the arctangent of  $X$  in radians.

Result is in the range  $-\pi/2$  to  $\pi/2$ . The expression  $X$  may be any numeric type, but the evaluation of ATN is always performed in single precision.

### Example

```
10 INPUT X
20 PRINT ATN(X)
RUN
?3
1.249046
Ok
```

## CDBL

### Syntax

CDBL ( X )

### Comments

Converts  $X$  to a double precision number.

### Example

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
454.67 454.6700134277344
Ok
```

## **CHR\$**

### **Syntax**

CHR\$( / )

### **Comments**

Returns a string whose one element has ASCII code *I*.

CHR\$(*nnn*) is commonly used to send a special character to the terminal. Replace *nnn* with the decimal ASCII code. (ASCII codes are listed in Appendix H.)

For example, the BEL character CHR\$(7) could be sent as a preface to an error message, or a form feed CHR\$(12) could be sent to clear a CRT screen and return the cursor to the home position.

### **Example**

```
PRINT CHR$(66)
B
Ok
```

### **Reference**

See the ASC function for ASCII-to-numeric conversion.

## CINT

### Syntax

CINT(X)

### Comments

Convert  $X$  to an integer by rounding the fractional portion.

If  $X$  is not in the range -32768 to 32767, an Overflow error occurs.

### Example

```
PRINT CINT(45.67)
46
Ok
```

### Reference

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type.

See also the FIX and INT functions, both of which return integers.

## COS

### Syntax

COS(X)

### Comments

Returns the cosine of  $X$  in radians.

Calculations of COS(X) are performed in single precision.

### Example

```
10 X = 2*COS(.4)
20 PRINT X
RUN
1.842122
Ok
```



## CSNG

### Syntax

CSNG ( X )

### Comments

Converts *X* to a single precision number.

### Example

```
10 A# = 975.342184#
20 PRINT A#; CSNG(A#)
RUN
    975.342184 975.3422
Ok
```

### Reference

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

## CSRLIN

### Syntax

CSRLIN

### Comments

Returns the current line (or row) position of the cursor. The value returned is in the range 1 to 25.

POS(*I*) returns the column location of the cursor.

The cursor position is set using the LOCATE statement.

### Reference

LOCATE, POS, SCREEN

## **CVI, CVS, CVD**

### **Syntax**

CVI(2-byte string)  
CVS(4-byte string)  
CVD(8-byte string)

### **Comments**

Converts string values to numeric values.

Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

### **Example**

```
.  
.   
.   
70 FIELD #1,4 AS N$, 12 AS B$,...  
80 GET #1  
90 Y=CVS(N$)  
.   
.   
. 
```

### **Reference**

MKI\$, MKS\$, MKD\$  
Appendix A

## DATE\$

### Syntax

DATE\$ = *string expression*

*string expression* = DATE\$

### Comments

Sets or retrieves the current system date.

The date is kept in DATE\$ and can be referred to or changed. The date is held in the 10-letter form yyyy-mm-dd where y is year, m is month, and d is day. This date must be set in DOS before GWBASIC is started up. The date can be formatted in the following manner:

mm-dd-yy  
mm/dd/yy  
mm-dd-yyyy  
mm/dd/yyyy

The year range is 1980 to 2077. If only one digit is used for the month or day, a zero (0) is placed in front of it. Furthermore, if the date is formatted with 2 digits, 19yy or 20yy is set automatically.

## ENVIRON\$

### Syntax

ENVIRON\$ [ *string parameter* ]  
ENVIRON\$ [ *n* ]

### Comment

Retrieves a parameter string from GWBASIC's Environment String Table.

The string result returned by ENVIRON\$ may not exceed 255 characters. If a parameter name is specified and either cannot be found or has no text following it, a null string is returned.

When the parameter name is specified, ENVIRON\$ returns all the associated text following *string parameter* in the Environment String Table.

If the argument *n* is used, the *n*th string in the Environment String Table is returned. It includes all text, and the parameter name.

If *n*th string doesn't exist, a null string is returned.

## EOF

### Syntax

EOF *file number*

### Comments

Tests to see if the file specified in *file number* has ended. *File number* is specified by the filespec in the OPEN statement.

The EOF function returns -1 (true) if the end of a sequential file has been reached. If true (-1) is returned for a communications file, this indicates that the buffer is empty.

Use EOF to test for end-of-file while inputting to avoid input past end errors.

## ERDEV and ERDEV\$

### Syntax

ERDEV  
ERDEV\$

### Comments

Provides a way to obtain device-specific status information.

ERDEV is an integer function that contains the error code returned by the last device to declare an error. ERDEV\$ is a string function that contains the name of the Device Driver that generated the error.

ERDEV is set by the Interrupt 24 (hex) handler when an error within MS-DOS is detected. ERDEV contains the INT 24 error code in the lower eight bits. Neither function can be set by the user.

### Example

If you have installed a device driver MYLPT2, and then run out of paper, the driver's error number for that problem is 9:

```
PRINT ERDEV, ERDEV$
```

yields:

```
9           MYLPT2
```

## **ERR and ERL**

### **Syntax**

*u* = ERR

*v* = ERL

### **Comments**

Returns the *error code* and *line number* associated with an error.

When an error handling subroutine is entered, the variable ERR contains the *error code* for the error, and the variable ERL contains the *line number* where the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL contains 65535. To test if an error occurred in a direct statement, use

```
IF 65535 = ERL THEN...
```

Otherwise, use

```
IF ERR = error code THEN...
```

or

```
IF ERL = line number THEN...
```

If the *line number* is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. GWBASIC error codes are listed in Appendix F.

## EXP

### Syntax

EXP(X)

### Comments

Returns  $e$  to the power of  $X$ .

$X$  must be  $\leq 88.02969$ . If EXP overflows, the Overflow error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.  $e$  is the base for natural logarithms.

### Example

```
10 X = 5
20 PRINT EXP (X-1)
RUN
  54.59815
Ok
```

## FIX

### Syntax

FIX(X)

### Comments

Returns the truncated integer part of  $X$ .

FIX(X) is equivalent to  $\text{SGN}(X) * \text{INT}(\text{ABS}(X))$ . The major difference between FIX and INT is that FIX does not return the next lower number for the negative  $X$ .

### Examples

```
PRINT FIX(58.75)
  58
Ok

PRINT FIX(-58.75)
 -58
Ok
```

## **FRE**

### **Syntax**

```
FRE ( 0 )  
FRE ( X$ )
```

### **Comments**

Returns the number of bytes in memory not being used by GWBASIC.

Arguments to FRE are dummy arguments. *FRE* ( " " ) causes GWBASIC to "clean house" before returning the number of free bytes. Housecleaning is when GWBASIC assembles all of the strings that are still in use, freeing up unused parts of memory for additional data. GWBASIC does not initiate garbage collection until all free memory has been used up. Therefore, using *FRE* ( " " ) periodically results in shorter delays for each housecleaning.

### **Example**

```
PRINT FRE(0)  
14542  
Ok
```

## **HEX\$**

### **Syntax**

```
HEX$ ( X )
```

### **Comments**

Returns a string which represents the hexadecimal value of the decimal argument. *X* is rounded to an integer before *HEX\$(X)* is evaluated.

### **Example**

```
10 INPUT X  
20 A$ = HEX$(X)  
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"  
RUN  
? 32  
32 DECIMAL IS 20 HEXADECIMAL  
Ok
```

## INKEY\$

### Syntax

INKEY\$

### Comments

Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal.

No characters are echoed and all characters are passed through to the program except for **CTRL BREAK** which terminates the program. See Appendix H, Extended Codes, for information on reading special keys.

### Example

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

## INP

### Syntax

INP ( / )

### Comments

Returns the byte read from port *I*. *I* must be in the range 0 to 65535. INP is the complementary function to the OUT statement.

### Example

```
100 A=INP(54321)
```

In assembly language, this is equivalent to:

```
MOV DX,54321
IN AL,DX
```



## INPUT\$

### Syntax

INPUT\$(X[,[#]Y])

### Comments

Returns a string of *X* characters, read from the terminal or from file number *Y*.

If the terminal is used for input, no characters are echoed and all control characters are passed through except **CTRL BREAK** which is used to interrupt the execution of the INPUT\$ function.

### Examples

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN HEXADECIMAL
10 OPEN"1",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
.
.
.
```

## INSTR

### Syntax

INSTR([ / ,] X\$, Y\$)

### Comments

Searches for the first occurrence of string *Y\$* in *X\$* and returns the position at which the match is found.

Optional offset *I* sets the position for starting the search. *I* must in the range 1 to 255. If *I*>LEN(*X*\$) or if *X*\$ is null or if *Y*\$ cannot be found, INSTR returns *I* or 1. *X*\$ and *Y*\$ may be string variables, string expressions, or string literals.

#### **Example**

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$); INSTR(4,X$,Y$)
RUN
  2  6
Ok
```

**Note:** If *I*≤0 is specified, error message Illegal function call in xxx is returned.

## **INT**

#### **Syntax**

INT(*X*)

#### **Comments**

Returns the largest integer ≤ *X*.

#### **Examples**

```
PRINT INT(99.89)
99
Ok
```

```
PRINT INT(-12.11)
-13
Ok
```

#### **Reference**

See the FIX and CINT functions, which also return integer values.

## IOCTL\$

### Syntax

IOCTL\$ ([#] *filenumber* )

### Comments

Receives acknowledgment that an IOCTL statement succeeded or failed, or obtains current status information.

IOCTL\$ could also be used to ask a communications device to return the current baud rate, information on the last error, logical line width, etc.

IOCTL\$ only works if:

1. The device driver is installed.
2. The device driver states it processes IOCTL strings.
3. GWBASIC performs an OPEN on a file on that device.

### Example

```
10 OPEN "\DEV\PHONE" AS #1
20 IOCTL #1, "RAW"
```

This example tells the device that the data is raw.

```
30 IF IOCTL$ (1) = "F" THEN CLOSE 1
```

In this situation, if the character driver PHONE responds with an F from the raw data mode IOCTL statement, then the file is closed.

### Reference

IOCTL

## **LEFT\$**

### **Syntax**

LEFT\$(X\$, /)

### **Comments**

Returns a string comprising the leftmost *I* characters of X\$.

*I* must be in the range 0 to 255. If / is greater than LEN(X\$), the entire string (X\$) is returned. If *I*=0, the null string (length zero) is returned.

### **Example**

```
10 A$ = "BASIC86 "  
20 B$ = LEFT$(A$,5)  
30 PRINT B$  
BASIC  
Ok
```

### **Reference**

MID\$, RIGHT\$

## **LEN**

### **Syntax**

LEN(X\$)

### **Comments**

Returns the number of characters in X\$. Non-printing character and blanks are counted.

### **Example**

```
10 X$ = "PORTLAND, OREGON"  
20 PRINT LEN(X$)  
16  
Ok
```

## LOC

### Syntax

`LOC(file number)`

### Comments

Returns the present location in the file.

With random disk files, LOC returns the record number just read or written. With sequential files, LOC returns the number of sectors (in 128 byte blocks) read from or written to the file since it was opened.

If a sequential file is opened in the input mode, GWBASIC reads the first sector and the LOC function value is set at 1.\* Numeric values given in sequential files are in 128 byte blocks.

For a communications file, this statement returns the number of characters in the input queue waiting to be read. The input queue can hold more than 255 characters (determined by the /C: switch). If there are more than 255 characters in the queue, LOC(x) returns 255. Since a string is limited to 255 characters, this practical limit alleviates the need for the programmer to test for the string size before reading the data into it. If fewer than 255 characters remain on the queue, LOC(x) returns the actual count.

LOC(*n*) functions for the disk files. For Random files, LOC returns the actual record position within the file.

For sequential files, LOC returns the current byte position in the file divided by 128. This provides compatibility with programs written for earlier versions or GWBASIC.

\*When a file is opened for APPEND or OUTPUT, LOC returns the size of the file in bytes divided by 128.

## LOF

### Syntax

LOF(*file number*)

### Comments

The LOF function returns the number of bytes allocated to the file.

*File number* is tied to a currently opened file.

LOF(*n*) functions for disk files. For Random files, LOF returns the size of the file in bytes.

For sequential files, LOF returns the size of the file in bytes. When a file is opened for APPEND or OUTPUT, LOF returns the size of the file in bytes.

When the disk file is assigned, the actual size of the file is returned in byte units. In GWBASIC, this file is made with units of 128 bytes, therefore the value is an integer multiple of 128.

When the communication file is specified, LOF returns the amount of free space in the input buffer. That is, /C:size-LOC(x). The buffer size is ordinarily 256, but it is possible to change it with the optional /C:switch.

### Example

```
100 OPEN "O", #1, "TEST.DAT"
110 INPUT A$
120 PRINT #1, A$
130 CLOSE #1
140 OPEN "I", #2, "TEST.DAT"
150 PRINT LOF(2)
160 CLOSE #2
170 END
```

## LOG

### Syntax

LOG ( X )

### Comments

Returns the natural logarithm of X. X must be greater than zero.

### Example

```
PRINT LOG(45/7)
1.860752
Ok
```

## LPOS

### Syntax

LPOS ( X )

### Comments

Returns the current position of the line printer print head within the line printer buffer. This does not necessarily give the physical position of the print head at the time that LPOS is called. X is a dummy argument.

### Example

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

## MID\$

### Syntax

MID\$(X\$, / [, J])

### Comments

Returns a string of length *J* characters from *X\$*, beginning with the *I*th character.

*I* and *J* must be in the range 1 to 255. If *J* is omitted or if there are fewer than *J* characters to the right of the *I*th character, all rightmost characters beginning with the *I*th character are returned. If *I*>LEN(*X\$*), MID\$ returns a null string.

### Example

```
LIST
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,8,8)
Ok
RUN
GOOD EVENING
Ok
```

**Note:** If *I*=0 is specified, error message Illegal argument in line number is returned.



## **MKI\$, MKS\$, MKD\$**

### **Syntax**

```
MKI$( integer expression )  
MKS$( integer expression )  
MKD$( integer expression )
```

### **Comments**

Convert numeric values to string values.

Any numeric value that is placed in a random file buffer with a LSET or RSET statement must be converted to a string. MKI\$ converts an integer to a 2-byte string. MKS\$ converts a single precision number to a 4-byte string. MKD\$ converts a double precision number to an 8-byte string.

### **Example**

```
90  AMT=(K+T)  
100 FIELD #1, 4 AS D$, 20 AS N$  
110 LSET D$ = MKS$(AMT)  
120 LSET N$ = A$  
130 PUT #1  
.  
.
```

### **Reference**

CVI, CVS, CVD  
Appendix B.

## **OCT\$**

### **Syntax**

```
OCT$( X )
```

### **Comments**

Returns a string that represents the octal value of the decimal argument.

*X* is rounded to an integer before OCT\$(*X*) is evaluated.

**Example**

```
PRINT OCT$(24)
30
Ok
```

**Reference**

See the HEX\$ function for hexadecimal conversion.

**PEEK****Syntax**

```
PEEK ( / )
```

**Comments**

Returns the byte (decimal integer in the range 0 to 255) read from memory location *I*, using the segment specified in the last DEF SEG statement.

*I* must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement.

**Example**

```
A=PEEK(&H5A00)
```

## **PLAY(*n*)**

### **Syntax**

$v = \text{PLAY} ( n )$

### **Comments**

Returns the number of notes currently in the background music queue.

*n* is a dummy argument and may be any value.

PLAY(*n*) returns 0 when in Music Foreground mode.

## **PMAP**

### **Syntax**

$v = \text{PMAP} ( \textit{expression}, \textit{function} )$

### **Comments**

Maps an expression to logical or physical coordinates as follows:

<i>function</i> = 0	Maps Logical expression to Physical x.
1	Maps Logical expression to Physical y.
2	Maps Physical expression to Logical x.
2	Maps Physical expression to Logical y.

## POINT

### Syntax

POINT( *x,y* )

### Comments

Reads the attribute value of a pixel from the screen.

(*x,y*) are coordinates of the point that is being investigated.

The color code of the dot with coordinates assigned by (*x,y*) is returned as a numerical value.

If the point given is out of range, the value -1 is returned.

The attribute value depends on the graphics mode:

320x200	0-3
640x200	0,1

## POINT (function)

### Syntax

POINT( *function* )

### Comments

Returns the value of the current x or y graphics accumulator as follows.

<i>function</i> = 0	Returns the current Physical x coordinate.
1	Returns the Current Physical y coordinate.
2	Returns the Current Logical x coordinate if WINDOW is active, else returns the Current Physical coordinate as in 0 above.
3	Returns the Current Logical y coordinate if WINDOW is active, else returns the Current Physical coordinate as in 1 above.

## POS

### Syntax

POS ( / )

### Comments

Returns the current cursor position.

The leftmost position is 1. *I* is a dummy argument.

### Example

```
IF POS(X)>60 THEN PRINT CHR$(13)
```

### Reference

LPOS

## RIGHT\$

### Syntax

RIGHT\$(X\$, / )

### Comments

Returns the rightmost *I* characters of string *X\$*.

*I* must be  $\geq 0$ . If  $I = \text{LEN}(X\$)$ , returns *X\$*. If  $I = 0$ , the null string (length zero) is returned.

### Example

```
10 A$="DISK GWBASIC"  
20 PRINT RIGHT$(A$,7)  
RUN  
GWBASIC  
Ok
```

### Reference

MID\$, LEFT\$

## RND

### Syntax

RND [( X )]

### Comments

Returns a random number between 0 and 1.

The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE). However,  $X < 0$  always restarts the same sequence for any given  $X$ .

$X > 0$  or  $X$  omitted generates the next random number in the sequence.  $X = 0$  repeats the last number generated.

### Example

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
12 65 86 72 79 12
```

## SCREEN

### Syntax

$x = \text{SCREEN}(\text{row}, \text{col} [, z])$

### Comments

Returns the ASCII code (0-255) for the character from the screen at the specified *row* (line) and *column*.

$x$  is a numeric variable receiving the ordinal returned.

*Row* is a valid numeric expression returning an unsigned integer in range 1 to 25.

*Col* is a valid numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending on the WIDTH.

$z$  is a valid numeric expression returning a boolean result.



## SIN

### Syntax

SIN(*X*)

### Action

Returns the sine of *X* in radians. SIN(*X*) is calculated in single precision:  $\text{COS}(X) = \text{SIN}(X + 3.14159/2)$

### Example

```
PRINT SIN (1.5)
.9974951
Ok
```

## SPACE\$

### Syntax

SPACE\$(*X*)

### Comments

Returns a string of spaces length *X*. The expression *X* is rounded to an integer and must be in the range 0 to 255.

### Example

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
1
2
3
4
5
Ok
```

### Reference

SPC



## SPC

### Syntax

SPC ( / )

### Comments

Prints *I* blanks on the terminal. SPC may only be used with PRINT and LPRINT statements.

*I* must be in the range 0 to 255. A semicolon (;) is assumed to follow the SPC(*I*) command.

### Example

```
PRINT "OVER" SPC(16) "THERE"  
OVER          THERE  
Ok
```

### Reference

SPACE\$

## SQR

### Syntax

SQR ( X )

### Comments

Returns the square root of *X*. *X* must be  $\geq 0$ .

### Example

```
10 FOR X = 10 TO 25 STEP 5  
20 PRINT X, SQR(X)  
30 NEXT  
RUN  
10          3.162278  
15          3.872984  
20          4.472136  
25          5  
Ok
```

## STICK

### Syntax

STICK(*n*)

### Comments

Returns the x and y coordinates of the two joysticks.

*n* is a numeric expression return an unsigned integer in the range of 0 to 3. The values are:

- 0 Returns the x coordinate for joystick A. Also stores the x and y values for both joysticks for the following function calls:
- 1 Returns the y coordinate of joystick A at the time of the last call to stick (0).
- 2 Returns the x coordinate of joystick B at the time of the last call to stick (0).
- 3 Returns the y coordinate of joystick B at the time of the last call to stick (0).

### Example

```
10 CLS
20 LOCATE 1,1
30 PRINT "X= ";STICK(0)
40 PRINT "Y= ";STICK(1)
50 GOTO 20
```

Creates an endless loop to display the value of the x,y coordinate for joystick A. You may use the **CTRL** and **BREAK** keys to terminate the endless loop.

## **STR\$**

### **Syntax**

STR\$

### **Comments**

Returns a string representation of the value of X.

### **Example**

```
5  REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N))-1 GOSUB 30,100,200,300,400,500
.
.
.
```

### **Reference**

VAL

## STRIG

### Syntax

$x = \text{STRIG}(n)$

### Comments

Return the status of a specified joystick trigger.

$n$  is a numeric expression returning an unsigned integer in the range 0 to 3, designating which trigger is to be checked.  $x$  is a numeric variable for storing the result of the function.

The values for returning  $n$  are:

- 0 Returns -1 if trigger A was pressed since the last STRIG(0) statement; 0 if not.
- 1 Returns -1 if trigger A is currently down; 0 if not.
- 2 Returns -1 if trigger B was pressed since the last STRIG(2) statement; 0 if not.
- 3 Returns -1 if trigger B is currently down; 0 if not.

When a joystick trap occurs, that occurrence of the event is destroyed. Therefore the  $x=\text{STRIG}(n)$  function always returns false inside a subroutine, unless the event has been repeated since the trap. If you want to perform different procedures for various joysticks, you must set up a different subroutine for each joystick, rather than a single subroutine for all procedures.

### Example

```
10 IF STRIG(0) THEN BEEP
20 GOTO 10
```

This is an endless loop which beeps whenever the trigger button on joystick 0 is pressed. You may use the **CTRL** and **BREAK** keys to terminate the endless loop.

## STRING\$

### Syntax

```
STRING$( / , J )  
STRING$( / , X$ )
```

### Comments

Returns a string of length *I* whose characters all have ASCII code *J* or the first character of *X\$*.

### Example

```
10 X$ = STRING$(10,45)  
20 PRINT X$ "MONTHLY REPORT" X$  
RUN  
MONTHLY REPORT  
Ok
```

## TAB

### Syntax

```
TAB( / )
```

### Comments

Spaces to position *I* on the terminal. If the current print position is already beyond space *I*, TAB goes to that position on the next line.

Space 1 is the leftmost position, and the rightmost position is the width of the output device. *I* must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

### Example

```
10 PRINT "NAME" TAB(25) "AMOUNT": PRINT  
20 READ A$,B$  
30 PRINT A$ TAB(25) B$  
40 DATA "G.T. JONES", "$25.00"  
RUN  
NAME                                AMOUNT  
  
G.T. JONES                          $25.00  
Ok
```

## **TAN**

### **Syntax**

`TAN ( X )`

### **Comments**

Returns the tangent of *X* in radians.

TAN(*X*) is calculated in single precision. If TAN overflows, the Overflow error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

### **Example**

```
10 Y = Q*TAN(X)/2
```

## **TIME\$**

### **Syntax**

`TIME$ = string expr`  
Sets the current time.

`string expr = TIME$`  
Sets the current time.

### **Comments**

Sets or retrieves the current time.

*String expr* is a valid string constant or variable.

TIME\$ is a system variable and has a built-in clock

For *string expr* = TIME\$, TIME\$ returns an 8-character string in the form *hh:mm:ss* where *hh* is the hour (00 to 23), *mm* is the minutes (00 to 59), and *ss* is the seconds (00 to 59).

For `TIME$ = string expr`, *string expr* may be one of the following forms:

1. *hh* sets the hour. Minutes and seconds default to 00.
2. *hh:mm* sets the the hour and minutes. Seconds default to 00
3. *hh:mm:ss* sets the hour, minutes, and seconds.

If any of the values are out of range, an illegal function call error appears.

If *string expr* is not a valid string, a Type mismatch error results.

### **Example**

```
100 INPUT "TIME(HH:MM:SS)=";T$
110 TIME$=T$
120 CLS
130 LOCATE 10,20
140 PRINT "PRESENT TIME IS ";TIME$
150 GOTO 130
```

## TIMER

### Syntax

$v = \text{TIMER}$

### Comments

Returns a single precision floating point number representing the elapsed number of seconds since midnight or system reset.

It includes the fractional seconds to the degree possible. `TIMER` is a read-only function. `TIMER` may not be used as a user variable.

### Example

```
RANDOMIZE TIMER
```

## USR

### Syntax

`USR [ digit ]( X )`

### Comments

Calls the user's assembly language subroutine with the argument *X*.

*Digit* is in the range 0 to 9 and corresponds to the digit supplied with the `DEF USR` statement for that routine. If *digit* is omitted, `USR0` is assumed.

### Example

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
.
.
.
```



## VAL

### Syntax

VAL ( X\$ )

### Comments

Returns the numerical value of string X\$.

The VAL function also strips leading blanks, tabs, and line feeds from the argument string. For example:

```
VAL (" -3 ")
```

returns -3.

### Example

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
   PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>90801 AND VAL(ZIP$)<=90815 THEN
   PRINT NAME$ TAB(25) "LONG BEACH"
.
.
.
```

### Reference

See the STR\$ function for numeric-to-string conversion.

## VARPTR

### Syntax

VARPTR(*variable name*)

VARPTR(*#file number*)

### Comments

Format 1 returns the address of the first byte of data identified with *variable name*. A value must be assigned to *variable name* prior to execution of VARPTR. Otherwise an illegal function call error results. Any type *variable name* may be used (numeric, string, array), and the address returned is an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(O)) is usually specified when passing an array, so that the lowest addressed element of the array is returned.

**Note:** All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Format 2 returns the starting address of the GWBASIC file control block (FCB) assigned to *file number*. (This is different from the MS-DOS file control block.)

VARPTR should be called just before you use it, because FCB is dynamically allocated during the execution of a program.

The FCB ( file control Block ) structure is as follows:

Location	Length	Description
0	1	The mode at file opening 1 Sequential input 2 Sequential output 4 Random access 16 APPEND 32 Reserved by the system 128 Reserved by the system
1	38	MS-DOS file control block
39	2	For sequential files: the number of sectors accessed For random files: the accessed record number +1
41	1	Pointer to indicate the sector byte currently accessed
42	1	Number of bytes remaining in the input buffer
43	3	Reserved by the system
46	1	Device number 0,1 Diskette drives A: and B: 248 LPT3 249 LPT2 250 COM2 251 COM1 253 LPT1 254 SCN: 255 KYBD: 252 COM3:
47	1	Output width
48	1	Location of the pointer in the buffer for PRINT#
49	1	Used by GWBASIC for LOAD and SAVE operations
50	1	Output location to be used for TAB extension
51	128	Physical data buffer to be used for data transfer between DOS and GWBASIC. This can also be used for a data test in sequential access.
179	2	The size of a variable-length record. The default value is 128 bytes. The value of <i>record length</i> in OPEN command is set.
181	2	Current physical record number
183	2	Current logical record number
185	1	Reserved by the system
186	2	Location of the input/output pointer for PRINT#, INPUT#, and WRITE#. This is used only for a disk file.
188	n	Field buffer. Size n is set by <i>record length</i> in OPEN command. This can be used to test data in the random access mode.

#### Example

```
100 X=USR(VARPTR(Y))
```

## VARPTR\$

### Syntax

VARPTR\$ (*variable*)

### Comments

Returns the address of a *variable* in a character form.

The address where *variable* data is stored is returned in 3-byte character format. Before executing VARPTR\$, a value should be assigned to the specified *variable*. The address for the *variable* is given in the following format:

Byte 0	Byte 1	Byte 2
Type	Low-order byte of variable address	High-order byte of variable address

The type, showing the type of *variable name*, is one of the following:

- 2 Integer type
- 3 Character type
- 4 Single-precision real type
- 8 Double-precision real type

Type value that VARPTR\$ returns is the same as

CHR\$(type)+MKI\$(VARPTR\$(variable name))

VARPTR\$ is mainly used to indicate the *variable* name in the PLAY and DRAW statements. For example, if we specify the following by a PLAY statement:

```
PLAY "XA$;"
```

Although this may be used as it is, it can also be written by using the VARPTR\$ Function:

```
PLAY "X" + VARPTR$(A$)
```



## APPENDIX A

### GW BASIC DISK I/O

Disk input/output (I/O) procedures for the beginning GW BASIC user are examined in this chapter. If you are new to GW BASIC or if you're receiving disk-related errors when running GW BASIC, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. The MS-DOS operating system appends the default extension .BAS to the filename given in a SAVE, RUN, MERGE, or LOAD command.

#### PROGRAM FILE COMMANDS

The following commands and statements are used in GW BASIC program file manipulation.

SAVE *filename* [,A]

Writes to disk the program that is currently residing in memory. Option A writes the program as a series of ASCII characters. Otherwise, GW BASIC uses a compressed binary format.

LOAD *filename* [,R]

Loads the program from disk into memory. LOAD always deletes the current contents of memory and closes all files before LOADING. The R option runs the program immediately. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files. (LOAD *filename*,R and RUN *filename*,R are equivalent.)

RUN *filename* [,R]

Loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN *filename*,R and LOAD *filename*,R are equivalent.)

MERGE *filename*

Loads the program from disk into memory but does not delete the the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the merged program resides in memory, and GWBASIC returns to command level.

KILL *filename*

Deletes the file from the disk. *Filename* may be a program file, a sequential or random access data file, or any other type of file.

NAME *old-filename* AS *new-filename*

To change the name of a disk file, execute the NAME statement. NAME may be used with any type of files.

## PROTECTED FILES

If you wish to save a program in an encoded binary format, use the Protect (,P) option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

## DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a GWBASIC program: sequential files and random access files.

### SEQUENTIAL FILES

Sequential files are easier to create than random files, but are limited in flexibility and speed when it comes to accessing the data. The data written into a sequential file is a series of ASCII characters that are stored, one after another (sequentially), in the order they are sent. This data is read back out of the file in the same way.

The statements and functions that are used with sequential files are:

OPEN	PRINT#	INPUT#	WRITE#
CLOSE	PRINT# USING	LINE INPUT#	INPUT\$
EOF	LOC	LOF	

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN "O",#1,"DATA"

Opens the file in O mode.

2. PRINT#1,A\$;B\$;C\$

Writes data to the file using the PRINT# statement. (WRITE# may be used instead.)

3. CLOSE #1  
OPEN "I",#1,"DATA"

To access the data in the file, you must CLOSE the file and reOPEN it in I mode.

4. INPUT#1,X\$,Y\$,Z\$

Use the INPUT# statement to read data from the sequential file into the program.

5. CLOSE #1

Close the file when done.



## Program 1 -- Creating a Sequential Data File

Program 1 creates a sequential file, DATA, from information you input at the terminal:

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN CLOSE #1:END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;" ";D$;" ";H$
60 PRINT:GOTO 20
RUN
```

```
NAME? MICKEY MOUSE
DEPARTMENT? CARTOONS
DATE HIRED? 01/12/72
```

```
NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65
```

```
NAME? EBENEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78
```

```
NAME? CLARK KENT
DEPARTMENT? FLIGHT INSTRUCTION
DATE HIRED? 08/16/78
```

```
NAME? etc.
```

## Program 2 -- Accessing a Sequential File

Now look at Program 2. It accesses the file DATA that was created in Program 1 and displays the name of everyone hired in 1978:

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEZER SCROOGE
CLARK KENT
Input past end in 20
Ok
```

Program 2 reads every item in the file, sequentially. When all the data has been read, line 20 causes an Input past end error. To avoid getting this error, insert line 15 below, which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement,

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of 128 byte records that have been written to or read from the file since it was OPENed.

## Adding Data to a Sequential File

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in O mode and destroy its current contents. Instead, you may open the file for APPEND.

## RANDOM FILES

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because GWBASIC stores them in a packed binary format, as a series of ASCII characters.

The biggest advantage to random files is that data can be accessed randomly (i.e., anywhere on the disks). It is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in numbered units called *records*.

The statements and functions that are used with random files are:

OPEN	FIELD	LSET	RSET
PUT	CLOSE	LOC	LOF
MK1\$	MKS\$	MKD\$	GET
CVS	CVI	CVD	

### Creating a Random File

The following program steps are required to create a random file:

1. OPEN "R",#1,"FILE",32

OPEN the file for random access (R mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

2. FIELD #1,20 AS N\$,4 AS A\$, 8 AS P\$

Use the FIELD statement to allocate space in the random buffer for the variables that are written to the random file. Note that in most cases the sum of the lengths of the variables should be equal to the record length, and in no case should it be greater than the record length.

3. LSET N\$=X\$  
LSET A\$=MKS\$(AMT)  
RSET P\$=TEL\$

Use LSET or RSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKS\$ for a single precision value, and MKD\$ for a double precision value.

4. PUT #1,CODE%

Write the data from the buffer to the disk using the PUT statement.

5. CLOSE #1

Close the file when all desired information has been written.

### Program 3 -- Creating a Random File

Program 3 takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. Each record consists of a 20-character name, an amount which is a single precision number, and an 8-character phone number. The two-digit code that is input in line 30 becomes the record number. If 999 is entered, the program closes the file and ends.

**Note:** Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE (OR 999 TO END)";CODE%
40 IF CODE%=999 THEN CLOSE #1:END
50 INPUT "NAME";X$
60 INPUT "AMOUNT";AMT
70 INPUT "PHONE";TEL$:PRINT
80 LSET N$=X$
90 LSET A$=MKS$(AMT)
100 RSET P$=TEL$
110 PUT #1,CODE%
120 GOTO 30
```

### Accessing a Random File

The following program steps are required to access a random file:

1. OPEN "R",#1,"FILE",32

OPEN the file in R mode.

2. FIELD #1,20 AS N\$, 4 AS A\$, 8 AS P\$

Use the FIELD statement to allocate space in the random buffer for the variables that are read from the file.

**Note:** In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement. In this case, you do not need to close the file until all operations (input and output) on the file are done.

3. GET #1, CODE%

The GET statement moves the desired record to the random buffer.

4. PRINT N\$  
PRINT CVS(A\$)

The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

5. CLOSE #1

Close the file when done.

#### **Program 4 -- Accessing A Random File**

Program 4 accesses the random file FILE that was created in Program 3. By inputting the two-digit code at the terminal, the information associated with that code is read from the file and displayed:

```
10 OPEN "R", #1, "FILE", 32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE"; CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##"; CVS(A$)
70 PRINT P$: PRINT
80 GOTO 30
```

The LOC function, with random files, returns the current record number. The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1) > 50 THEN END
```

ends the program execution, if the current record number in file #1 is higher than 50.

#### **Program 5 -- Inventory**

Program 5 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory contains no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR\$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```

120 OPEN "R", #1, "INVEN.DAT", 39
125 FIELD #1, 1 AS F$, 30 AS D$, 2 AS Q$, 2 AS R$, 4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1, "INITIALIZE FIELD"
140 PRINT 2, "CREATE A NEW ENTRY"
150 PRINT 3, "DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4, "ADD TO STOCK"
170 PRINT 5, "SUBTRACT FROM STOCK"
180 PRINT 6, "DISPLAY ALL ITEMS BELOW REORDER LEVEL"
190 PRINT 7, "END"
220 PRINT:PRINT:INPUT "FUNCTION";FUNCTION
225 IF (FUNCTION<1) OR (FUNCTION>7) THEN PRINT
    "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
235 IF FUNCTION=7 THEN CLOSE #1:END
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT "OVERWRITE";A$:
    IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUALITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT #1, PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$$.#";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT #1, PART%
550 RETURN
560 REM SUBTRACT FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)-S%

```

```

620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;
    " IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
    "REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF ASC(F$)=0 THEN IF CVI(Q$)<(R$) THEN
    PRINT D$;" QUANTITY";CVI(Q$) TAB(50)
    "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF (PART%<1) OR (PART%>100) THEN
    PRINT "BAD PART NUMBER":GOTO 840 ELSE
    GET#1,PART%:RETURN
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN

```

## APPENDIX B

### COMMUNICATIONS

In this appendix, the GWBASIC statements necessary to implement communication through the RS-232C interface are described.

#### 1. Communication file

The communication file is opened by OPEN COM and the input/output buffer is allocated to it. It is closed by CLOSE.

#### 2. Communication I/O unit

A communication unit is opened like a file. All input/output statements that are valid for disk files are valid for communications. Sequential input statements are the same for disk files and communications. These are:

```
INPUT# file number  
LINE INPUT# file number  
INPUT$
```

The communication sequential output statements are also the same for disk files. These are:

```
PRINT# file number  
PRINT# file number USING  
WRITE# file number
```

#### 3. GET and PUT statements communication

GET and PUT statements are different for communications files and disk files. They are used for fixed-length records I/O to and from the communication file. Instead of specifying the record number to be read or written, you specify the number of bytes to be transmitted to or received from the file buffer. The value should not exceed the LEN option for the OPEN COM statement.



#### 4. I/O function

Difficulties with asynchronous communication occur when processing is performed at "character" speed. At high baud rates (1200 and up), it is often necessary to suspend character transmission from another computer to prevent the data buffer from overflowing. This can often be done by sending XOFF (CHR\$(19)) to the remote computer, and XON CHR\$(17)) when ready to start transmission again. XOFF tells the remote computer to stop sending. XON tells it to restart transmission.

**Note:** Other codes may be used if they do not overlap with data. This depends on the protocol you are using to communicate with the remote system.

GWBASIC provides three types of functions to detect the correct input conditions:

LOC(*x*)

Number of characters in the input buffer to be read. If it is greater than 255, 255 is returned.

LOF(*x*)

Returns the number of characters free in the input buffer. This value is equal to  $n - \text{LOC}(x)$ .  $n$  is the size of the communication buffer set by the GWBASIC command /C: option. The default for  $n$  is 256.

EOF(*x*)

Is true (-1) if characters are in the input buffer, false (0) if the buffer is empty.

**Note:** If a read is attempted when the input buffer is full, or 0 is returned by LOF (file number), a Communication buffer overflow error occurs.

#### 5. INPUT\$ function

When reading communication files, INPUT\$ is more convenient than INPUT# and LINE INPUT# statements. With communication, all the ASCII characters are valid. The INPUT# statement terminates when a comma or carriage return is encountered. The LINE INPUT# statement terminates when a carriage return is found.

With INPUT\$, all characters are assigned to strings. INPUT\$(x,#y) means that x characters are returned from file #y. the following statements are most efficient for reading a communication file:

```
10 WHILE NOT EOF (1)
20 B$=INPUT$(LOC(1),#1)
.
.      (Data in B$ is processed.)
.
100 WEND
```

This subprogram returns the characters in the communication buffer into B\$. The middle statements process them as strings. If there are more than 255 characters, the first 255 characters in the buffer are processed at one time. This prevents String overflow errors from occurring.

## 6. Device Errors

An I/O error will sometimes be generated upon opening of the COM file or when the first input is done. If this occurs in a program, the ON ERROR GOTO statement should be used to trap the error and retry the operation.



## APPENDIX C

### GW BASIC ASSEMBLY LANGUAGE SUBROUTINES

GW BASIC has provisions for interfacing with assembly language subroutines via the USR function and the CALL statement.

The USR function allows assembly language subroutines to be called in the same way GW BASIC intrinsic functions are called. However, the CALL statement is the recommended way of interfacing 8086 machine language programs with GW BASIC. It is compatible with more languages than is the USR function call, it produces more readable source code, and it can pass multiple arguments.

### MEMORY ALLOCATION

**Note:** Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory location minus the amount of memory needed for the assembly language subroutine(s) with the /M: switch.

In addition to the GW BASIC interpreter code area, GW BASIC uses up to 64K of memory beginning at its data segment (DS).

If, when an assembly language subroutine is called, more stack space is needed, GW BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. GW BASIC's stack must be restored, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the operating system or the GW BASIC POKE statement.

### The CALL Statement

As mentioned earlier, the CALL statement is the recommended way of interfacing machine language programs with GW BASIC. It is further suggested that the old style user-call USR(n) not be used. The syntax is:

CALL *variable name* [( *argument list* )]

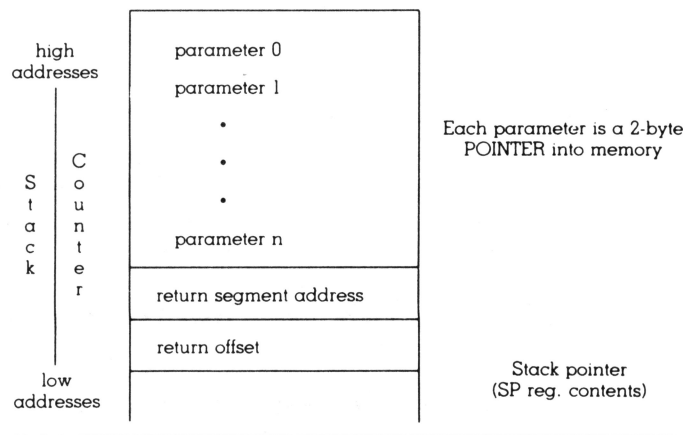
*Variable name* contains the segment offset that is the starting point in memory of the subroutine being CALLED.

*Argument list* contains the variables or constants, separated by commas, that are to be passed to the routine.

Invoking the CALL statement caused the following to occur:

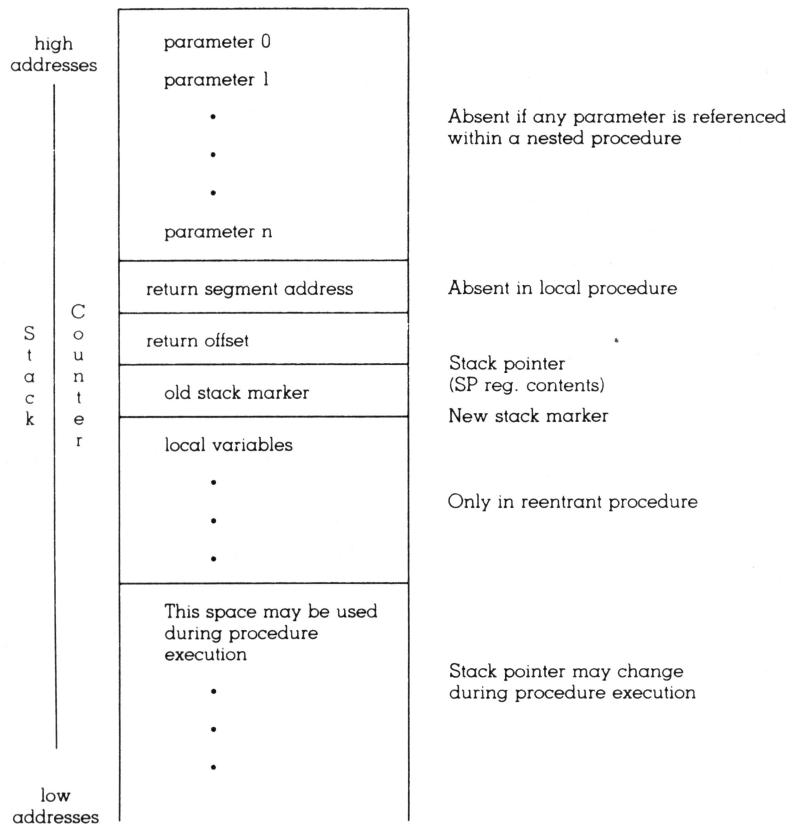
1. For each parameter in the argument list, the 2 byte offset of the parameter's location within the data segment (DS) is pushed onto the stack.
2. GWBASIC's return address code segment (CS), and offset (IP) are pushed onto the stack.
3. Control is transferred to the user's routine using the segment address given in the last DEF SEG statement and the offset given in *variable name*.

These actions are illustrated by the two following diagrams. Figure C-1 illustrates the state of the stack at the time of the CALL statement. Figure C-2 illustrates the condition of the stack during execution of the CALLED subroutine.



**Figure C-1** Stack Layout when CALL Statement Is Activated

The user's routine now has control. Parameters may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to (BP).



**Figure C-2** Stack Layout during Execution of a CALL Statement

You must observe the following rules when coding a subroutine:

1. The CALLED routine may destroy the AX, BX, CX, DX, SI, DI, and BP registers.
2. The CALLED program MUST know the number and length of the parameters passed. References to parameters are positive offsets added to (BP) (assuming the called routine moved the current stack pointer into BP; i.e., MOV BP,SP). That is, the location of p1 is at 8(BP), p2 is at 6(BP), p3 is at 4(BP), and so on.
3. The CALLED routine must do a RET *n* (where *n* is two times the number of parameters in the argument list) to adjust the stack to the start of the calling sequence.
4. Values are returned to GWBASIC by including in the argument list the variable name that receives the result.

5. If the argument is a string, the parameter's offset points to 3 bytes called the *string descriptor*.

Byte 0 of the string descriptor contains the length of the string (0 to 255) Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

**Note:** If the argument is a string constant in the program, the string descriptor points to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add "+" to the string constant in the program. For example:

```
20 A$ = "GWBASIC"+" "
```

This forces the string constant to be copied into string space. Now the string may be modified without affecting the program.

6. Strings may be altered by user routines, but the length must not be changed. GWBASIC cannot correctly manipulate strings if their lengths are modified by external routines.

### Examples

```
100 DEF SEG=&H8000
110 F=&H7FA
120 CALL F(A,B$,C)
```

Line 100 sets the segment to 8000 Hex. The value of variable F is added into the address as the low word after the DEF SEG value is left shifted 8 bits. (This is a function of the microprocessor, not of GWBASIC.) Here, F is set to &H7FA, so that the call to F executes the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

The following sequence of assembly language demonstrates success of the parameters passed and storing a return result in the variable 'C'.

```
MOV     BP,SP           ;Get current stack position in BP.
MOV     BX,6[BP]        ;Get address of B$ information.
MOV     CL,[BX]         ;Get length of B$ in CL.
MOV     DX,1[BX]        ;Get address of B$ text in DX.
.
.
.
MOV     SI,8[BP]        ;Get address of 'A' in SI.
MOV     DI,4[BP]        ;Get pointer to 'C' in DI.
MOVS    WORD            ;Store variable 'A' in 'C'.
RET     6               ;Restore stack, return.
```

**Note:** The called program must know the variable type for numeric parameters passed. In the above example, the instruction

MOVS WORD

copies only 2 bytes. This is fine if variables A and C are integer. GWBASIC would have to copy 4 bytes if they were single precision and copy 8 bytes if they were double precision.

## USR FUNCTION CALLS

Although the CALL statement is the recommended way of calling assembly language subroutines, the USR function call is still available for compatibility with previously-written programs.

The syntax of the USR function call is:

USR[ *digit* ][( *argument* )]

*Digit* is from 0 to 9. *Digit* specifies the USR routine being called (see DEF USR). If *digit* is omitted, USR0 is assumed.

*Argument* is any numeric or string expression, enclosed in parentheses. Arguments are discussed in detail below.

A DEF SEG statement MUST be executed prior to USR call to assure that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine. (See Chapter 2, DEF SEG.)

For each USR function, a corresponding DEF USR statement must have been executed to define the USR call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register AL contains a value which specifies the type of argument that was given. The value in AL may be one of the following:

- |   |  |
|---|--|
| 2 | Two-byte integer (two's complement)    |
| 3 | String                                 |
| 4 | Single precision floating point number |
| 8 | Double precision floating point number |

If the argument is a number, the BX register points to the 8-byte long Floating Point Accumulator (FAC) area where the argument is stored.



If the argument is an integer:

- FAC-2            Contains the upper 8 bits of the argument.
- FAC-3            Contains the lower 8 bits of the argument.

If the argument is not an integer:

- FAC              Contains the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.
- FAC-1            Contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

If the argument is a single precision floating point number:

- FAC-2            Contains the middle 8 bits of mantissa.
- FAC-3            Contains the lowest 8 bits of mantissa.

If the argument is a double precision floating point number:

- FAC-7            Contain four more bytes of mantissa.
- FAC-4            Contains the lowest 8 bits.

If the argument is a string, the DX register points to 3 bytes called the string descriptor. Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in GWBASIC's Data Segment.

**Note:** If the argument is a string constant in the program, the string descriptor points to program text. Be careful not to alter or destroy your program this way. See the CALL statement above. Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

### Examples

```
110     DEF USRO=&H8000        'Assumes user gave /M:32767
120     X=5                    'Note that X is single precision
130     Y = USRO(X)
140     PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed. Alternatively, it may be set to integer by calling FRCINT, a routine contained in GWBASIC, to set the BX register to the integer equivalent of the argument and then calling another routine, MAKINT, to pass the value in the BX register back to GWBASIC.

The following sample assembly language routine simply multiplies the argument passed by 2 and returns an integer result. Always be sure that your programs are defined by a PROC FAR statement.

DOUBLE	SEGMENT		
	ASSUME	CS:DOUBLE	
FRCINTOFFSET	EQU	103H	
MAKINTOFFSET	EQU	107H	
FRCINT	LABEL	DWORD	
	DW	FRCINTOFFSET	
FRCSEG	DW	?	
MAKINT	LABEL	DWORD	
	DW	MAKINTOFFSET	
MAKSEG	DW	?	
USRPRG	PROC	FAR	
	POP	SI	
	POP	AX	;Recover GWBASIC's CS
	PUSH	AX	
	PUSH	SI	
	MOV	FRCSEG,AX	;Set segment for long
			;indirect CALL
	MOV	MAKSEG,AX	
	CALL	FRCINT	;Force arg in FAC to int
			;in BX
	ADD	BX,BX	;BX = BX * 2
	CALL	MAKINT	;Put Result back in FAC
	RET		;Long return to GWBASIC
USRPRG	ENDP		
DOUBLE	END		

When FRCINT or MAKINT is called, and when the subroutine terminates with a return, ES, DS, and SS must have the same value they had when the subroutine was entered. These registers point to GWBASIC's Data Segment.



## APPENDIX D

### EXAMPLE OF EXECUTION

#### START GWBASIC

A>BASIC

When the MS-DOS prompt appears, type **basic** and press **RETURN**.

GW-BASIC 3.11  
(C) Copyright Microsoft 1983,1984,1985

The start message and the number of free bytes is displayed and GWBASIC starts.

Compatibility Software GW-BASIC V3.11  
Copyright (c) 1984, 1985 by Phoenix Software Associates Ltd.  
INCLUDES SUPPORT FOR EXTENDED VIDEO MODES  
VERSION 1.0 01-27-86

61955 Bytes free  
Ok

Ok indicates command level of GWBASIC.

```
10 PRINT"*** START ***"  
20 PRINT"* | *","* |+| *","* |x| *"  
30 FOR I=1 TO 10  
40 PRINT I,|+|,I*I  
50 PRINT "*** END ***"  
60 END
```

Make source program.

```
RUN  
*** START ***  
* | *      * |+| *      * |x| *  
FOR Without NEXT in 30  
Ok
```

Execute program in interpreter mode.  
The current program starts.

Error message.  
Return to command level, awaiting correction of error.

```
45 NEXT I  
RENUM  
Ok
```

Add line number 45 and renumber lines with **RENUM** command.

```

LIST
10 PRINT "*** START ***"
20 PRINT "* | *", "* |+| *", "* |x| *"
30 FOR I=1 TO 10
40 PRINT I, I+I, I*I
50 NEXT I
60 PRINT "*** END ***"
70 END
Ok

```

Display revised program with  
LIST command.

```

RUN
*** START ***
* | *      * |+| *      * |x| *
1          2          1
2          4          4
3          6          9
4          8         16
5         10         25
6         12         36
7         14         49
8         16         64
9         18         81
10        20        100
*** END ***
Ok

```

Execute in interpreter mode.

```

SAVE "TEST"
Ok

```

Save program as filename TEST.BAS.

```

FILES "TEST.BAS"
TEST .BAS
Ok

```

Check with FILES command.

```

SYSTEM

```

Return to MS-DOS with  
system command.

```

A>DIR TEST.BAS
TEST  BAS    128    1-01-80    12:17a
    1 File(s)

```

Check with DIR command.

```

A>CHKDSK

```

Check status of use of  
disk with CHKDSK command.

```

362496 bytes total disk space
25600 bytes in 3 hidden files
87040 bytes in 16 user files
249856 bytes available on disk

```

```

262144 bytes total memory
236776 bytes free

```

A>basic

Start GWBASIC.

GW-BASIC 3.11

(C) Copyright Microsoft 1983,1984,1985

The start message and the number of free bytes is displayed and GWBASIC starts again.

Compatibility Software GW-BASIC V3.11

Copyright (c) 1984, 1985 by Phoenix Software Associates Ltd.

INCLUDES SUPPORT FOR EXTENDED VIDEO MODES

VERSION 1.0 01-27-86

61955 Bytes free

Ok

Ok indicates return to command level.

LOAD"TEST"

Ok

Load TEST.BAS file

LIST

10 PRINT"\*\*\* START \*\*\*"

20 PRINT"\* | \*","\* |+| \*","\* |x| \*"

30 FOR I=1 TO 10

40 PRINT I,I+I,I\*I

50 NEXT I

60 PRINT"\*\*\* END \*\*\*"

70 END

Ok

Display program with LIST command.

EDIT 30

30 FOR I=1 TO 10

Revise line number 30 with EDIT command.

30 FOR I=1 TO 20

Ok

Type 20 to replace 10 and press RETURN.

LIST

10 PRINT"\*\*\* START \*\*\*"

20 PRINT"\* | \*","\* |+| \*","\* |x| \*"

30 FOR I=1 TO 20

40 PRINT I,I+I,I\*I

50 NEXT I

60 PRINT"\*\*\* END \*\*\*"

70 END

Ok

Display revised program with LIST command.

RUN

\*\*\* START \*\*\*

*   *	*   +   *	*   x   *
1	2	1
2	4	4
3	6	9
4	8	16
5	10	25
6	12	36
7	14	49
8	16	64
9	18	81
10	20	100
11	22	121
12	24	144
13	26	169
14	28	196
15	30	225
16	32	256
17	34	289
18	36	324
19	38	361
20	40	400

\*\*\* END \*\*\*

Ok

SAVE "TEST"

Ok

FILES "TEST.BAS"

TEST .BAS

Ok

RESET

Ok

SYSTEM

Execute program in  
interpreter mode.

Save program as TEST.BAS. This  
overwrites the previously saved file.

Check with FILES command.

Rewrite directory information  
information on disk using  
RESET command.

Exit to MS-DOS.

## APPENDIX E

### CONVERTING PROGRAMS TO LEADING EDGE GWBASIC

If you have a program which was originally written in another version of GWBASIC, some minor adjustments may be necessary before running them with this version. Here are some specific things to look for when converting to this version of the GWBASIC program.

#### STRING DIMENSION

Delete all statements that are used to declare the length of strings. A statement such as `DIM A$(I,J)`, which dimensions a string array for J elements of length I, should be converted to this GWBASIC statement:

```
DIM A$(J)
```

Some versions of BASIC use a comma or ampersand (&) for string concatenation. Each of these must be changed to a plus sign (+), which is the operator for GWBASIC string concatenation.

In Leading Edge GWBASIC, the `MID$`, `RIGHT$`, and `LEFT$` functions are used to take substrings of strings. Forms such as `A$(I)` to access the Ith character in A\$, or `A$(I,J)` to take a substring of A\$ from position I to position J, must be changed as follows:

Other BASIC	Leading Edge GWBASIC
<code>X\$=A\$(I)</code>	<code>X\$=MID\$(A\$,I,1)</code>
<code>X\$=A\$(I,J)</code>	<code>X\$=MID\$(A\$,I,J-I+1)</code>

Table E-1 Conversion of Substring References I



If the substring reference is on the left side of an assignment and X\$ is used to replace characters in A\$, convert as follows:

Other BASIC	Leading Edge GWBASIC
A\$(I)=X\$	MID\$(A\$,I,1)=X\$
A\$(I,J)=X\$	MID\$(A\$,I,J-I+1)=X\$

**Table E-2** Conversion of Substring References II

### **MULTIPLE ASSIGNMENTS**

Some versions of BASIC allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to 0. Leading Edge GWBASIC interprets the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

### **MULTIPLE STATEMENTS**

Some versions of BASIC use a backslash (\) to separate multiple statements on a line. With Leading Edge GWBASIC, multiple statements must be separated by a colon (:).

### **MAT FUNCTIONS**

Programs using the MAT functions to do matrix operations in some versions of BASIC must be rewritten using FOR...NEXT loops to execute properly.

**APPENDIX F**  
**SUMMARY OF ERROR CODES AND ERROR MESSAGES**

---

<b>Number</b>	<b>Message</b>
---------------	----------------

---

<b>1</b>	<b>NEXT without FOR</b>
----------	-------------------------

A variable in a NEXT statement does not correspond to any previous executed, unmatched FOR statement variable.

Adjust the program to match a FOR with a NEXT.

<b>2</b>	<b>Syntax error</b>
----------	---------------------

A line is encountered that contains some incorrect sequence of characters (such as unmatched parentheses, misspelled command or statement, incorrect punctuation, etc.)

Correct the line in error.

<b>3</b>	<b>RETURN without GOSUB</b>
----------	-----------------------------

A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.

Correct the program by adding a STOP or END statement to keep the computer from executing the subroutine portion of the program.

<b>4</b>	<b>Out of data</b>
----------	--------------------

A READ statement is executed when there are no DATA statements with unread data remaining in the program.

Refer to the section of this guide that covers the function found on that line of the program and correct the program.

---

**Number    Message**

---

**5        Illegal function call**

A parameter that is out of range is passed to a math or string function. This error may also occur as the result of:

- o    a negative or unreasonably large subscript
- o    a negative or zero argument with LOG
- o    a negative argument to SQR
- o    a negative mantissa with a non-integer exponent
- o    a call to a USR function for which the starting address has not yet been given
- o    an improper argument to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING\$, SPACES\$, INSTR, or ON...GOTO.

**6        Overflow**

The result of a calculation is too large to be represented in GWBASIC's number format. If underflow occurs, the result is zero and execution continues without an error.

Use smaller numbers, or change to single or double precision variables.

**7        Out of memory**

A program is too large, has too many FOR loops or GOSUBS, too many variables, or expressions that are too complicated.

At the beginning of your program, use CLEAR to set aside more memory or stack space.

**8        Undefined line**

A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.

Check your line numbers and correct the program.

**9        Subscript out of range**

An array element is referenced either with a subscript that is outside the dimension of the array, or with the wrong number.

Check how your array variables are used.

**10       Duplicate definition**

Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

Correct the program to only define each array once, or use the OPTION BASE statement before you encounter your arrays in the program.

---

**Number    Message**

---

**11      Division by zero**

A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

The program continues to run after this error is encountered. You need not do anything to correct this problem.

**12      Illegal direct**

A statement that is illegal in direct mode is entered as a direct mode command.

Enter that statement only a part of a program.

**13      Type mismatch**

A string variable name is assigned a numeric value or vice verse; a function that expects a numeric argument is given a string argument or vice versa.

Check your variables and correct this problem.

**14      Out of string space**

String variables have caused GWBASIC to exceed the amount of free memory remaining. GWBASIC allocates string space dynamically, until it runs out of memory.

**15      String too long**

An attempt is made to create a string more than 255 characters long.

Make smaller strings.

**16      String formula too complex**

A string expression is too long or too complex.

The expression should be broken into smaller expressions.

**17      Can't continue**

An attempt is made to continue a program that;

1. has halted due to an error,
2. has been modified during a break in execution, or
3. does not exist.

If the program is loaded into memory, use the RUN command to start it.

---

**Number    Message**

---

- 18      **Undefined user function**  
A USR function is called before the function definition (DEF FN statement) is given.  
  
Use DEF FN before the function is encountered in the program.
- 19      **No RESUME**  
An error trapping routine is entered but contains no RESUME statement.  
  
Include RESUME in the error trapping routine.
- 20      **RESUME without error**  
A RESUME statement is encountered before an error trapping routine is entered.  
  
Include a STOP or END statement before the subroutines section of your program is reached.
- 21      **Unprintable error**  
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.
- 22      **Missing operand**  
An expression contains an operator with no operand following it.  
  
Make sure your expression is complete.
- 23      **Line buffer overflow**  
An attempt is made to input a line that has too many characters.  
  
Use more than one line for multiple statements and use string variables in place of constants, if appropriate.
- 24      **Device Time-out**  
GWBASIC program cannot get data or ready signal in regular time from device.  
  
Retry the operation.
- 25      **Device Fault**  
Hardware (printer, disk, etc.) is out of order.  
  
Check your device and retry.
- 26      **FOR without NEXT**  
A FOR was encountered without a matching NEXT.  
  
Correct your program.

---

Number	Message
--------	---------

---

- |    |   |
|----|---|
| 27 | <p><b>Out of paper</b></p> <p>Printer is in unusual condition of paper empty, no power, or no connection.</p> <p>Check your printer and retry.</p>  |
| 29 | <p><b>WHILE without WEND</b></p> <p>A WHILE statement does not have a matching WEND.</p> <p>Correct your program.</p>   |
| 30 | <p><b>WEND without WHILE</b></p> <p>A WEND was encountered without a matching WHILE.</p> <p>Correct your program.</p>   |
| 50 | <p><b>Field overflow</b></p> <p>A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.</p> <p>Check that the OPEN and FIELD statements match up.</p>  |
| 51 | <p><b>Internal error</b></p> <p>An internal malfunction has occurred in GWBASIC.</p> <p>Recopy your disk. If the problem persists, report to your computer dealer the conditions that caused this message to appear.</p>  |
| 52 | <p><b>Bad file number</b></p> <p>A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.</p> <p>Check that the file you want is open and that you've assigned the correct file number, and that you have a valid file specification.</p> |
| 53 | <p><b>File not found</b></p> <p>A LOAD, KILL, or OPEN statement reference a file that does not exist on the current disk.</p> <p>Check your disk and file specifier, then retry the operation.</p>  |
| 54 | <p><b>Bad file mode</b></p> <p>An attempt is made to use PUT, GET or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mod other than I, O, or R.</p> <p>Is the OPEN statement included and proper? PUT and GET statements require a random file.</p>   |

---

**Number    Message**

---

- 55      File already open**  
A sequential output mode OPEN is issued for a file that is already open or a KILL is given for a file that is open.
- Make sure you use only one OPEN for each file. A file must be closed before it can be KILLED.
- 57      Device I/O error**  
An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.
- Restart your computer.
- 58      File already exists**  
The filename specified in a NAME statement is identical to a filename already in use on the disk.
- Use a different name and retry this operation.
- 61      Disk full**  
All disk storage is in use.
- Erase any unneeded files on the active disk or use a new diskette.
- 62      Input past end**  
An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file.
- Use the EOF function to detect the end of the file.
- 63      Bad record number**  
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.
- Correct your program.
- 64      Bad file name**  
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).
- Correct your file specification and try again.
- 66      Direct statement in file**  
A direct statement is encountered while LOADING an ASCII-format file. The LOAD is terminated.
- Correct your program. Each statement must have a related line number.

Number	Message
67	<p><b>Too many files</b></p> <p>An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.</p> <p>Use a new disk and try again.</p>
68	<p><b>Device unavailable</b></p> <p>An attempt was made to open the file for an unavailable device or the specified I/O device is unavailable.</p> <p>Check to be sure the device is properly installed.</p>
69	<p><b>Communication buffer overflow</b></p> <p>The data transmitted from the communication port overflowed in the input buffer.</p> <p>Correct your program to use an ON ERROR statement and review the possible causes of this problem for a solution.</p> <ol style="list-style-type: none"> <li>1. Increase the size of your communications buffer with the /C: option.</li> <li>2. Use another protocol between the machines.</li> <li>3. Send and receive at a slower rate.</li> </ol>
70	<p><b>Disk write protect</b></p> <p>An attempt was made to write data to a write protected diskette.</p> <p>Check your diskette. If you want to write onto this diskette, remove the write protect tab.</p>
71	<p><b>Disk not ready</b></p> <p>The disk drive door is open or the drive is empty.</p> <p>Correct the situation and retry.</p>
72	<p><b>Disk medium error</b></p> <p>An abnormal condition occurred on the disk drive or diskette. Usually, this is caused by scratches on the diskette.</p> <p>Copy your files to a new diskette. Reformat the disk producing this error. If formatting fails, discard the disk.</p>
74	<p><b>Rename across disks</b></p> <p>An attempt was made to rename a file that was declared to be on a disk other than the disk specified for the old name. The renaming operation is not performed.</p>



---

**Number    Message**

---

**75       Path/file access error**

During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to make a correct Path to File name connection. The operation is not completed.

**76       Path not found**

During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to find the path specified. The operation is not completed.

**—       Can't continue after SHELL**

Upon returning from a child process, the SHELL statement finds insufficient memory to continue. GWBASIC closes all open files and exits to MS-DOS.

**—       You cannot SHELL to BASIC**

During initialization, GWBASIC discovers that it is being run as a child task. GWBASIC terminates and returns control to the parent copy of GWBASIC.

## APPENDIX G

### MATHEMATICAL FUNCTIONS

#### DERIVED FUNCTIONS

Functions that are not intrinsic to GWBASIC may be calculated as follows.

Function	GWBASIC Equivalent
SECANT	$\text{SEC}(X)=1/\text{COS}(X)$
COSECANT	$\text{CSC}(X)=1/\text{SIN}(X)$
COTANGENT	$\text{COT}(X)=1/\text{TAN}(X)$
INVERSE SINE	$\text{ARCSIN}(X)=\text{ATN}(X/\text{SQR}(-X*X+1))$
INVERSE COSINE	$\text{ARCCOS}(X)=\text{ATN}(X/\text{SQR}(-X*X+1))+1.5708$
INVERSE SECANT	$\text{ARCSEC}(X)=\text{ATN}(X/\text{SQR}(X*X+1))$ $+ \text{SGN}(\text{SGN}(X)-1)*1.5708$
INVERSE COSECANT	$\text{ARCCSC}(X)=\text{ATN}(X/\text{SQR}(X*X-1))$ $+ (\text{SGN}(X)-1)*1.5708$
INVERSE COTANGENT	$\text{ARCCOT}(X)=\text{ATN}(X)+1.5708$
HYPERBOLIC SINE	$\text{SINH}(X)=(\text{EXP}(X)-\text{EXP}(-X))/2$
HYPERBOLIC COSINE	$\text{COSH}(X)=(\text{EXP}(X)+\text{EXP}(-X))/2$
HYPERBOLIC TANGENT	$\text{TANH}(X)=\text{EXP}(-X)/\text{EXP}(X)+\text{EXP}(-X)*2+1$
HYPERBOLIC SECANT	$\text{SECH}(X)=2/(\text{EXP}(X)+\text{EXP}(-X))$
HYPERBOLIC COSECANT	$\text{CSCH}(X)=2/(\text{EXP}(X)-\text{EXP}(-X))$
HYPERBOLIC COTANGENT	$\text{COTH}(X)=\text{EXP}(-X)/\text{EXP}(X)-\text{EXP}(-X)*2+1$
INVERSE HYPERBOLIC SINE	$\text{ARCSINH}(X)=\text{LOG}(X+\text{SQR}(X*X+1))$
INVERSE HYPERBOLIC COSINE	$\text{ARCCOSH}(X)=\text{LOG}(X+\text{SQR}(X*X-1))$
INVERSE HYPERBOLIC TANGENT	$\text{ARCTANH}(X)=\text{LOG}(1+X)/(1-X)/2$
INVERSE HYPERBOLIC SECANT	$\text{ARCSECH}(X)=\text{LOG}((\text{SQR}(-X*X+1)+1)/X)$
INVERSE HYPERBOLIC COSECANT	$\text{ARCCSCH}(X)=\text{LOG}((\text{SGN}(X)*\text{SQR}(X*X+1)+1)/X)$
INVERSE HYPERBOLIC COTANGENT	$\text{ARCCOTH}(X)=\text{LOG}((X+1)/(X-1))/2$



## APPENDIX H

### ASCII Character Codes

ASCII codes have special meanings in GWBASIC. They are listed in Table H-1 below:

Decimal	Hex	Meaning in GWBASIC 2.11
007	07	beep
009	09	tab
010	0A	line feed
011	0B	home
012	0C	form feed
028	1C	cursor right
029	1D	cursor left
030	1E	cursor up
031	1F	cursor down

Table H-1 Special ASCII Codes in GWBASIC

### ASCII CODES

Table H-2 lists the ASCII codes (in decimal format with the corresponding ASCII character). These characters are printable by using the CHR\$(x) function, where *x* is the decimal ASCII code.

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	A	097	a
002	●	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	♦	EOT	036	\$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	■	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♪	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	▲	DLE	048	0	080	P	112	p
017	▼	DC1	049	1	081	Q	113	q
018	↕	DC2	050	2	082	R	114	r
019	!!	DC3	051	3	083	S	115	s
020	π	DC4	052	4	084	T	116	t
021	\$	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↑	ETB	055	7	087	W	119	w
024	↕	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	☐

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
128	Ç	160	á	192	Ł	224	α
129	ù	161	í	193	ł	225	β
130	é	162	ó	194	ŧ	226	Γ
131	ä	163	ü	195	Ƨ	227	π
132	ā	164	ñ	196	—	228	Σ
133	à	165	Ñ	197	+	229	σ
134	ä	166	g	198	ƒ	230	μ
135	ç	167	q	199	ƒ	231	τ
136	é	168	¿	200	ℓ	232	Φ
137	ë	169	¿	201	ℓ	233	θ
138	è	170	¿	202	ℓ	234	Ω
139	ì	171	½	203	ℓ	235	6
140	ì	172	¼	204	ℓ	236	×
141	ì	173	ì	205	=	237	Ø
142	À	174	«	206	≠	238	€
143	À	175	»	207	≠	239	⌋
144	É	176	■	208	≠	240	⌋
145	æ	177	■	209	≠	241	⌋
146	Æ	178	■	210	≠	242	⌋
147	ó	179	—	211	≠	243	⌋
148	ó	180	—	212	≠	244	⌋
149	ó	181	—	213	≠	245	⌋
150	ù	182	—	214	≠	246	⌋
151	ù	183	—	215	≠	247	⌋
152	ÿ	184	—	216	≠	248	⌋
153	Ö	185	—	217	≠	249	⌋
154	Ü	186	—	218	≠	250	⌋
155	é	187	—	219	≠	251	√
156	£	188	—	220	≠	252	n
157	₣	189	—	221	≠	253	2
158	Pt	190	—	222	≠	254	■
159	f	191	—	223	≠	255	(blank 'FF')

Table H-2 ASCII Codes 0-255

## EXTENDED CODES

For a combined key input or a special key input, the INKEY\$ variable returns an extended code. This code consists of two bytes. The first byte is for &H00 and the second byte is for one of the codes shown below. The two-byte code is received by INKEY\$. If the first byte is found to contain &H00, determine the entered key by the code in the second byte:

Second Code	Meaning
3	(Null character) NUL
15	(Shift tab) <b>SHIFT TAB</b>
16-25	<b>ALT</b> plus Q, W, E, R, T, Y, U, I, O, P
30-38	<b>ALT</b> plus A, S, D, F, G, H, J, K, L
44-50	<b>ALT</b> plus Z, X, C, V, B, N, M
59-68	Function keys <b>F1</b> through <b>F10</b> (when disabled as soft keys)
71	<b>HOME</b>
72	(Cursor Up)
73	(Page Up) <b>PGUP</b>
75	(Cursor Left)
77	(Cursor Right)
79	<b>END</b>
80	(Cursor Down)
81	(Page Down) <b>PGDN</b>
82	(Insert) <b>INS</b>
83	(Delete) <b>DEL</b>
84-93	<b>SHIFT F1</b> through <b>F10</b>
94-103	<b>CTRL F1</b> through <b>F10</b>
104-113	<b>ALT F1</b> through <b>F10</b>
114	<b>CTRL PRTSC</b>
115	(Previous Word) <b>CTRL</b>
116	(Next Word) <b>CTRL</b>
117	<b>CTRL END</b>
118	<b>CTRL PGDN</b>
119	<b>CTRL HOME</b>
120	<b>ALT</b> plus 1, 2, 3, 4, 5, 6, 7, 8, 9, -, =
132	<b>CTRL PGUP</b>

**Table H-3** Extended Codes



## **APPENDIX I**

### **SYNTAX LIST**

The typical form of commands, statements, functions, and system variables are described here for easy reference. For more detail, refer to Chapter 2 and Chapter 3.

#### **COMMANDS**

1. File
2. Program Execution
3. Program Creation
4. Output to Printer
5. Return to MS-DOS

#### **STATEMENTS**

1. Non I/O
2. Screen
3. Music
4. Input/Output Terminal
5. Port
6. Input/Output File
7. Function Key
8. Communication
9. Error Handling
10. Debugging
11. Pen Handling

#### **FUNCTIONS AND VARIABLES**

1. Arithmetic
2. String
3. Input
4. File
5. Miscellaneous



## COMMANDS

### 1. File

- o **BLOAD *filename, offset***  
Loads a file anywhere in user memory.
- o **BSAVE *filename, offset, length***  
Saves portions of the user memory on the specified device.
- o **FILES**  
Displays filenames on the disk.
- o **KILL *filename***  
Deletes a file from disk.
- o **LOAD *filename***  
Loads a program file from disk into memory.
- o **MERGE *filename***  
Merges a program file into the program in memory.
- o **NAME *old file AS new filename***  
Changes the name of a disk file.
- o **RESET**  
Closes all disk files.
- o **SAVE *filename***  
Saves a program file on disk.

### 2. Program Execution

- o **CONT**  
Continues the program execution.
- o **RUN *filename***  
Executes the program currently in memory.

### 3. Program Creation

- o **AUTO *line number, increment***  
Generates a line number automatically.
- o **CLEAR *expression 1, expression 2***  
Sets all variables and stack to initial value.
- o **DELETE *line number - line number***  
Deletes program lines.

- o EDIT *line number*  
Enters Edit Mode at the specified line.
- o LIST *line number - line number, device*  
Lists a program to the screen.
- o NEW  
Deletes the program currently in memory and clear all variables.
- o RENUM *new number, old number, increment*  
Renumbers program lines.

#### 4. Output to Printer

- o LCOPY  
Prints contents of the screen.
- o LLIST *line number-line number*  
Lists all or part of the program currently in memory.

#### 5. Return to MS-DOS

- o SYSTEM  
Leaves BASIC, returning to MS-DOS.

### STATEMENTS

#### 1. Non I/O

- o CALL *variable name (argument list)*  
Calls the machine language subroutine.
- o CHAIN *filename*  
Calls a program and passes variables to it from the current program.
- o COMMON *list of variables*  
Passes variables to a CHAINED program.
- o DEF FN *name (parameter list) = function definition*  
Define and name a function.
- o DEF *type range of letters*  
Declares variable type as integer, single precision, double precision, or string.

- o **DEF SEG = *address***  
Defines the current segment of storage.
- o **DEF USR *digit* = *integer expression***  
Specifies the starting address of an assembly language subroutine.
- o **DIM *list of subscripted variables***  
Specifies the maximum values for array variable subscripts and allocates storage accordingly.
- o **END**  
Terminates program execution and closes open files.
- o **ERASE *list of array variables***  
Eliminates arrays from a program.
- o **FOR *variable* = *x* TO *y***  
Executes a series of statements to be performed a given number of times by a loop.
- o **GOSUB *line number***  
Branches to a subroutine.
- o **GO TO *line number***  
Branches unconditionally out of the normal program sequence to a specified line number.
- o **IF *expression* THEN *statements* ELSE *statements***  
Makes a decision regarding program flow based on the result returned by an expression.
- o **LET *variable* = *expression***  
Assigns the value of an expression to a variable.
- o **MID\$ ( *string exp 1*, *n*, *m* ) = *string exp 2***  
Replaces a portion of one string with another string.
- o **NEXT *variable***  
Specifies the end of a FOR loop.
- o **ON *expression* GOSUB *list of line numbers***  
Branches to one of several specified subroutines.
- o **ON *expression* to GOTO *list of line numbers***  
Branches to one of several specified line numbers.
- o **OPTION BASE *n***  
Declares the minimum value for array subscript.
- o **RANDOMIZE *expression***  
Reseeds the random number generator.

- o **REM *remark***  
Allows explanatory remarks to be inserted in a program.
- o **RETURN**  
Returns from a subroutine.
- o **SHELL *parameter***  
Executes a child process.
- o **STOP**  
Terminates program execution and return to command level.
- o **SWAP *variable, variable***  
Exchanges the value of two variables.
- o **WEND**  
Specifies the end of WHILE loop.
- o **WHILE *expression***  
Executes a series of statements in a loop as long as a given condition is true.

## 2. Screen

- o **CIRCLE (*xcenter, ycenter*), *radius, color***  
Draws a circle with a center and radius.
- o **CLS *screen identifier***  
Erases the current active screen page.
- o **COLOR *foreground, background, border***  
Sets colors on the text screen.
- o **COLOR *background, palette***  
Sets colors in the graphics mode.
- o **DRAW *command string***  
Draws figures on the screen.
- o **GET (*x1, y1*) - (*x2, y2*) *arrayname***  
Reads points from an area of the screen.
- o **LINE (*x1, y1*) - (*x2, y2*)**  
Draws straight lines and rectangles on the screen in graphics mode.
- o **LOCATE *row, col***  
Moves the cursor to the specified position on the active screen.
- o **LOCATE *row, col***  
Moves the cursor to the specified position on the active screen.
- o **PAINT (*xstart, ystart*), *point attribute, border attribute***  
Fills in an arbitrary graphics figure of the specified color.

- o **PRESET (*x, y*)**  
Draws a dot at the assigned position on the screen in the background color.
- o **PSET (*x, y*)**  
Draws a dot at the assigned position on the screen in the designated color.
- o **PUT (*x, y*), *array name*, *operation***  
Displays graphics patterns at the assigned position on the screen.
- o **SCREEN *mode* (*,burst*) (*,active page*) (*,visual page*)**  
Sets screen attributes for either text or graphic displays.
- o **WIDTH *size***  
Sets the printed width in number of characters for the screen (or line printer).

### 3. Music

- o **BEEP**  
Sounds the speaker at approximately 1000 Hz for 1/4 second.
- o **PLAY *string expression***  
Plays a melody indicated by character string.
- o **SOUND *freq, duration***  
Generates sound through the speaker.

### 4. Input/Output (Terminal)

- o **INPUT "*prompt*"; *list of variables***  
Inputs from the terminal during program execution.
- o **LINE INPUT "*prompt*"; *string variable***  
Inputs an entire line to a string variable.
- o **LPRINT *list of expressions***  
Prints data at the line printer.
- o **LPRINT USING *string exp*; *list of expressions***  
Prints data at the line printer.
- o **PRINT *list of expressions***  
Outputs data at the terminal.
- o **PRINT USING *string exp*; *list of expressions***  
Prints strings of numbers using a specified format.
- o **WRITE *list of expressions***  
Outputs data at the terminal.

## 5. Port

- o **OUT *port, data***  
Sends a byte to a machine output port.
- o **POKE *port, data***  
Writes a byte into memory location.
- o **WAIT *port, n, m***  
Suspends program execution while monitoring the status of a machine input port.

## 6. Input/Output (File)

- o **CLOSE # *file number***  
Concludes I/O to a disk file.
- o **FIELD # *file number, field width* AS *string variable***  
Allocates space for variables in a random file buffer.
- o **GET # *file number, record number***  
Reads a record from a random disk file into a random buffer.
- o **INPUT # *file number, variable list***  
Reads data items from a sequential disk file and assigns them to program variables.
- o **LINE INPUT# *file number, string variable***  
Reads an entire line from a sequential disk data file to a string file.
- o **LSET *string variable* = *string expression***  
Moves data from memory to a random file buffer (left-justifies).
- o **OPEN *file mode 1, # file number, file spec, reclen***  
Opens a file.
- o **OPEN *filename* FOR *file mode 2* AS # *file number***  
Alternative form for OPEN.
- o **PRINT #*file number, list of expressions***  
Writes data to a sequential disk file.
- o **PRINT #*file number* ,USING *string exp;list of exps***  
Writes data to a sequential disk file.
- o **PUT # *file number, record number***  
Writes a record from a random buffer to a random disk file.
- o **RSET *string variable* = *string expression***  
Moves data from memory to a random file buffer (right-justifies)
- o **WRITE # *file number, list of expressions***  
Writes data to a sequential disk file.

## 7. Function Key

- o KEY *key number, string expression*  
Defines the function key assignment text.
- o KEY ON/OFF/LIST  
Displays function keys and turns display off.
- o KEY (*key number*) ON/OFF/STOP  
Enables (or disables) interrupts caused by a specified key.
- o ON KEY *key number* GOSUB *line number*  
Defines the starting line of the subroutine used when a KEY interrupt occurs.

## 8. Communications

- o COM(*n*) ON/OFF/STOP  
Enables and disables the trapping of communications.
- o ON COM(*n*) GOSUB *line number*  
Defines the starting line of the subroutine used when data arrives at the communication buffer.
- o OPEN "COM *n*: *options*" AS # *file number*  
Opens a communications file.

## 9. Error Handling

- o ERROR *integer expression*  
Simulates the occurrence of a BASIC error; or allows error codes to be defined by the user.
- o ON ERROR GOTO *line number*  
Enables error trapping and specifies the first line of the error handling subroutine.
- o RESUME *line number* /NEXT/O  
Continues program execution after an error recovery procedure has been performed.

## 10. Debugging

- o TRON, TROFF  
Enables (or disables) tracing of the execution of program statements.

## 11. Pen Handling

- o **ON PEN GOSUB *line number***  
Defines the starting line of the subroutine employed when the light pen is used.
- o **PEN ON/OFF/STOP**  
Enables (or disables) the light pen.

## FUNCTIONS AND VARIABLES

### 1. Arithmetic

- o **ABS (X)**  
Returns the absolute value of the expression X.
- o **ATN (X)**  
Returns the arctangent of X in radians
- o **CDBL (X)**  
Converts X to a double precision number.
- o **CINT (X)**  
Converts X to an integer by rounding the fractional portion.
- o **COS (X)**  
Returns the cosine of X in radians.
- o **CSNG (X)**  
Converts X to a single precision number.
- o **EXP (X)**  
Returns e to the power of X.
- o **FIX (X)**  
Returns the truncated integer part of X.
- o **INT (X)**  
Returns the largest integer  $\leq X$ .
- o **LOG (X)**  
Returns the natural logarithm of X.
- o **RND (X)**  
Returns a random number between 0 and 1.
- o **SGN (X)**  
Returns a value depending on the sign of X.



- o **SIN (X)**  
Returns the sine of X in radians.
- o **SQR (X)**  
Returns the square root of X.
- o **TAN (X)**  
Returns the tangent of X in radians.

## 2. String Handling

- o **ASC (X\$)**  
Returns a numerical value that is the ASCII code of the first character of the string X\$.
- o **CHR\$ (X)**  
Returns a string whose one element has ASCII code X.
- o **CVI (X\$), CVS (X\$), CVD (X\$)**  
Converts a string value to a numerical value.
- o **HEX\$ (X)**  
Returns a string that represents the hexadecimal value of the decimal argument.
- o **INSTR (I, X\$, Y\$)**  
Searches for the first occurrence of string Y\$ in X\$ and returns the position where the match is found.
- o **LEFT\$ (X\$, I)**  
Returns a string comprising the left-most I characters of X\$.
- o **LEN (X\$)**  
Returns the number of characters in X\$.
- o **MID\$ (X\$, I, J)**  
Returns a string of length J characters from X\$ beginning with the Ith characters.
- o **MKI\$ (X), MKS\$ (X), MKD\$ (X)**  
Converts a numerical value to a string value.
- o **OCT\$ (X)**  
Returns a string which represents the octal value of the decimal argument.
- o **RIGHTS (X\$, I)**  
Returns the current cursor position.
- o **SPACES (X)**  
Returns a string of spaces of length X.

- o **STR\$ (X)**  
Returns a string representation of the value of X.
- o **STRING\$ (I, X\$)**  
Returns a string of length I whose characters all have the first character of X\$.
- o **VAL (X\$)**  
Returns the numerical value of string X\$.
- o **VARPTR\$ (variable)**  
Returns the address of a variable in a character form.

### 3. Input

- o **INKEY\$**  
Returns either a one-character string containing a character read from the terminal or a null string.
- o **INPUT\$ (X, #Y)**  
Returns a string of X characters, read from the terminal or from file number Y.

### 4. File Status

- o **EOF (*file number*)**  
Tests to see if the file specified in file number has ended.
- o **LOC (*file number*)**  
Returns the present location in the file.
- o **LOF (*file number*)**  
Returns the number of bytes allocated to the file.

### 5. Miscellaneous

- o **CSRLIN**  
Returns the current line (or row) position of the cursor.
- o **DATES**  
Retrieves the date in calendar form.
- o **DATE\$ = *string expression***  
Sets the date in calendar form.
- o **ERL**  
Returns the line number of the last error.

- o ERR  
Returns the error code for the last error.
- o FRE  
Returns the number of bytes in memory not being used by BASIC.
- o INP (X)  
Returns the byte read from port X.
- o LPOS (X)  
Returns the current position of the line printer print head within the line printer buffer.
- o PEEK (X)  
Returns the byte read from memory location I.
- o PEN (X)  
Reads the light pen.
- o POINT (X, Y)  
Returns the attribute value of a pixel from the screen.
- o POS (I)  
Returns the current cursor.
- o SCREEN (*row, col, z*)  
Returns the ASCII code (0-255) for the character from the screen at the specified row (line) and column.
- o SPC (X)  
Prints X blanks on the terminal.
- o TAB (X)  
Spaces to position X on the terminal.
- o TIMES\$  
Returns the present time in character form.
- o TIMES\$ = *string expression*  
Sets the time in character form.
- o USR *digit* (X)  
Calls the specified assembly language subroutine.
- o VARPTR (*variable name*)  
Returns the address of the variable.
- o VARPTR (*# file number*)  
Returns the starting address of the File Control Block assigned to file number.

**APPENDIX J**  
**KEY SCAN CODES**

<b>Key</b>	<b>Hexadecimal Scan Code</b>
<b>Function Keys</b>	
F1	3B
F2	3C
F3	3D
F4	3E
F5	3F
F6	40
F7	41
F8	42
F9	43
F10	44
<b>Numeric Keys</b>	
1	02
2 @	03
3 #	04
4 \$	05
5 %	06
6 ^	07
7 &	08
8 *	09
9 (	0A
0 )	0B

Key	Hexadecimal Scan Code
<b>Numeric Keypad</b>	
1 END	4F
2	50
3 PGDN	51
4	4B
5	4C
6	4D
7 HOME	47
8	48
9 PGUP	49
0 INS	52
. DEL	53
PRTSC *	37
-	4A
+	4E
<b>Alphabetic Keys</b>	
A	1E
B	30
C	2E
D	20
E	12
F	21
G	22
H	23
I	17
J	24
K	25
L	26
M	32
N	31
O	18
P	19
Q	10
R	13
S	1F
T	14
U	16
V	2F
W	11
Y	15
Z	2C

Key		Hexadecimal Scan Code
<b>Punctuation Keys</b>		
\		2B
- _		0C
= +		0D
[ {		1A
] }		1B
` ~		29
; :		27
' "		28
, <		33
. >		34
/ ?		35
<b>Cursor Control Keys</b>		
TAB		0F
BACKSPACE		0E
SPACEBAR		39
<b>Control Keys</b>		
Left SHIFT		2A
NUM LOCK		45
SCROLL LOCK/BREAK		46
RETURN		1C
Right SHIFT		36
CAPS LOCK		3A
<b>Special Function Keys</b>		
ESC		01
CTRL	1D	
ALT		3B



## INDEX

### A

ABS 3-2, 3-11  
Absolute  
    form 1-30  
    value 3-2  
ADD\$ 2-37  
Address 2-23  
Algebraic expressions 1-19  
ALL option 2-7, 2-18  
Alphabetic characters 1-11  
ALT key 1-32, 1-33, 2-79  
AND 1-22, 2-115, 2-143  
Apage 2-129  
APPEND 2-86, 3-18, 3-19  
Arithmetic  
    operations 1-11  
    operators 1-19  
Arrays 2-42, 3-39  
    space 1-17  
    variables 1-16, 1-17, 2-97  
ASC 3-2, 3-4  
ASCII  
    codes 1-24, 3-27, H-1  
    format 2-63, 2-70, 2-128  
    mode 2-62  
Assignments, multiple E-2  
ATN 3-3  
ATP 1-2  
Attribute value 3-25  
AUTO 1-6, 2-2

### B

Background 2-14, 2-15  
BACKSPACE key 1-2, 1-32, 1-36  
BASIC  
    assembly language subroutines  
        C-1, C-5  
    \D 1-2

disk I/O A-1  
ending 1-1  
\F 1-2, 1-3  
\M 1-3  
machine language subroutines  
    1-3  
program editor 1-34  
program line format 1-11  
start options 1-2  
starting 1-1

Batch 1-2  
BEEP 2-3  
BEL 3-4  
BLOAD 2-4, 2-5  
Border 1-28, 2-14, 2-142  
Brackets, square Preface-3  
BSAVE 2-4, 2-5  
Buffer 1-2, 2-122  
    random file A-6  
    \C:size 1-2, 1-3  
    \C:nnn 1-2  
Burst 2-129

### C

Calculator functions 1-11  
CALL 2-4, 2-5, 2-6, 2-23, C-2  
Capital letters Preface-3  
CDBL 3-3, 3-5, 3-6  
CHAIN 2-7, 2-18, 2-91  
Character(s),  
    alphabetic 1-11, 1-25  
    numeric 1-11, 1-25  
    special 1-11, 1-12, 1-25, 3-4  
    type declaration 1-16  
CHDIR 2-9  
CHR\$ 2-2, 3-2, 3-4  
CINT 3-5, 3-6, 3-15  
CIRCLE 2-10, 2-59, 2-93  
CLEAR 2-11, 2-93



- CLOSE 2-12, 2-122
- CLS 2-13
- Codes,
  - ASCII character H-1
  - error F-1
  - extended H-3
  - key scan J-1
- COLOR 1-28, 2-10, 2-29, 2-59, 2-96, 2-112
  - display 1-28
  - (graphics) 2-16
  - (text) 2-14
- COM 2-17, 2-75, 2-78, 2-90
  - device 2-90
  - OFF 2-17, 2-75
  - ON 2-17, 2-75
  - STOP 2-17, 2-75
- COMMAND 2-131
- COMMAND.COM 2-131
- Command mode 1-5, 1-10, 1-11
- Commands, list of I-2
- Comments 2-1
- COMMON 2-7, 2-18
- Communications B-1
  - file B-1
  - I/O unit B-1
- Comspec 2-131
- Config.sys 1-3
- Constant(s), 1-14
  - double precision 1-15
  - fixed point 1-14, 2-20
  - floating point 1-14, 2-20
  - hex 1-14
  - integer 1-14, 2-20
  - numeric 1-14, 2-20
  - octal 1-14
  - single precision 1-15
  - string 1-14, C-5, C-6
- CONT 2-19, 2-61, 2-135
- Conversion, types 1-17
- Converting programs E-1
- Coordinates 1-30
- Correction, character (see Screen editor)
- Correction, line-by-line 1-7, 1-8
- COS 1-2, 3-5
- CRT screen 3-4
- CSNG 3-5, 3-6
- CSRLIN 1-28, 3-6
- CTRL key 1-33, 1-35
- CTRL-ALT-DEL keys 1-34, 2-143
- CTRL-BREAK keys 1-4, 1-6, 1-7, 1-34, 1-36 2-2, 2-19, 3-13, 3-14
- CTRL-END keys 1-9, 1-36
- CTRL-HOME keys 1-35, 1-36
- CTRL-NUM LCK keys 1-7, 1-34
- CTRL-Z keys 1-5

- Current node 1-27
- Cursor position 3-6
- CVD 3-7, 3-22
- CVI 3-7, 3-22
- CVS 3-7, 3-22

D

- DATA 2-20, 2-118, 2-119, 2-123
  - segment 2-23
- DATE\$ 3-8
- Debugging 1-11, 2-19, 2-141
- DEF FN 2-21
- DEF SEG 2-4, 2-5, 2-6, 2-23
  - address 2-23
- DEF USR 2-6, 2-24, 3-37
  - digit 2-24
- Default extension (.BAS) 1-2, 2-66, 2-70, 2-127
- DEFDBL 2-22
- DEFINT 2-22
- DEFSNG 2-22
- DEFSTR 2-22
- DEL key 1-36
- DELETE 1-8, 2-25
- Delimiters 1-12
- Derived functions G-1
- Device name 1-26
- Digit 2-24
- DIM 2-26
- Dimension, string E-1
- Direct mode 1-10, 2-76
- Directory
  - current node 1-27
  - paths 1-27
  - root node 1-27
- Disk data files A-2, A-5
- Display, monochrome 1-28
- Display, 16-color 1-28
- Division by zero 1-20
- Double precision 1-2
  - constant 1-15
  - conversion 1-17
  - number 3-5
  - space requirements 1-17
  - variable 1-16
- Double Precision Transcendental Math Package 1-2
- DRAW 2-27, 3-41
- Drive 1-25
- Dummy arguments 3-12

## E

EDIT 1-9, 2-30, 2-128  
 Ellipses Preface-3  
 ELSE 2-46  
 Embedded reserved words 1-16  
 END key 1-35, 1-36  
 END 2-12, 2-31, 2-44, 2-135  
 ENVIRON 2-32  
 ENVIRON\$ 2-33, 3-8  
 EOF 3-9  
 ERASE 2-34  
 ERDEV 3-9  
 ERDEV\$ 3-9  
 ERL 2-121, 3-10  
 ERR 3-10  
 ERROR 2-35, 2-75, 2-78  
 Error code 3-10, F-1  
 Error message(s) 1-24, F-1  
   Bad file mode 2-70  
   Can't continue after SHELL  
     2-133  
   Communication buffer overflow  
     1-18, B-2  
   Device I/O 2-89  
   Device timeout 2-90  
   Direct mode 2-76  
   Division by zero 1-20  
   Field overflow 2-37, 2-113  
   File already open 2-56  
   File not found 2-87  
   Illegal function call 2-15,  
     2-16, 2-23, 2-52, 2-67, 2-77,  
     2-95, 2-99, 2-109, 2-112,  
     2-116, 2-121, 2-128, 2-130,  
     3-2, 3-28, 3-36, 3-39  
   Input past end 3-9  
   NEXT without FOR 2-39, 2-40  
   Out of data 2-118  
   Out of memory 2-32, 2-93,  
     2-132  
   Overflow 3-5, 3-35  
   Redimensioned array 2-34  
   Redo from start 2-48  
   RESUME without error 2-124  
   Subscript out of range 2-26  
   Syntax 2-5, 2-76, 2-118  
   Type mismatch 2-21, 2-137,  
     3-36  
   Undefined line 2-46  
   Undefined line number 2-30  
   Undefined line xxxxx in yyyy  
     2-121  
   Undefined user function 2-21  
   Unprintable error 2-35  
   WEND without WHILE 2-144  
   WHILE without WEND 2-144  
 ESC key 1-6, 1-36

Examples 2-1

Exit 2-131

EXP 1-2, 3-11

Expression(s) 1-18

  algebraic 1-19

  arithmetic 1-18

  functional 1-18

  integer 2-35, 3-1

  logical 1-18

  numeric 2-85, 3-1

  relational 1-18

  string 3-1

  value 2-77

Extended codes H-3

Extension 1-25

  default (.bas) 1-2, 2-66,  
     2-70, 2-127

## F

FCB 1-2, 3-39, 3-40

FIELD 2-37, 2-86, A-7

  width 2-37

Fields

  numeric 2-107

  string 2-106

File(s) 1-25

  AUTOEXEC.BAT 2-133

  communication B-1

  input 1-4

  output 1-4

  random A-6, A-7

  random buffer 2-117

  random disk 3-18

  sequential 3-18

File Control Block (FCB) 1-2,  
   3-39, 3-40

Filemode 2-86

Filename 1-2, 1-25, 2-10

Filenumber 2-12

FILES 2-38

Filespec 1-25, 2-4, 2-7, 2-38

FIX 3-5, 3-11, 3-15

Fixed point constant 1-14

Floating point

  computation 2-47

  constant 1-14

  converted 1-18

  value 3-1

FOR 2-125

FOR...NEXT 2-39

Foreground 2-14

Form,

  absolute 1-30

  relative 1-30

FRE 3-12

Function keys 1-33

Functional operators 1-24

## Functions

- definition 2-21
- derived G-1
- EOF 1-5
- intrinsic 1-24, 3-1
- I/O B-2
- list of . I-1
- mathematical G-1
- transcendental 3-1
- user defined 1-24

## G

- GET 2-37, 2-41, 2-87, 2-90,  
2-115, 2-116, B-1  
(graphics) 2-42
- GML 2-27
- GOSUB 2-44, 2-120, 2-125
- GOSUB...RETURN 2-44, 2-125
- GOTO 2-19, 2-44, 2-45, 2-46,  
2-120, 2-121
- Graphics 2-115  
mode 1-28, 1-29
- GW BASIC, see BASIC
  - assembly language subroutines  
C-1
  - \D 1-2
  - disk I/O A-1
  - ending 1-1
  - \F 1-2
  - \M 1-3
  - machine language subroutines  
1-3
  - program editor 1-34
  - program line format 1-11
  - start options 1-2
  - starting 1-1

## H

### Hex

- constant 1-14
- scan codes J-1
- value 3-12
- HEX\$ 3-12, 3-23
- High resolution 1-29, 1-30,  
2-145
- HOME key 1-35
- Housecleaning 3-12

## I

- IF 2-46
- IF...THEN 2-46, 3-10
- Indirect mode 1-10

## Infinite

- loop 2-143
- width 2-146
- Infinity, machine 3-11
- INKEY\$ 2-78, 3-13
- INP 3-13
- INPUT 2-19, 2-37, 2-48, 2-87,  
2-92, A-7
- INPUT# 2-41, 2-50, 2-87, 2-89,  
2-150, B-2
- INPUT\$ 2-78, 3-14, B-2
- INS key 1-36
- INSTR 3-14, 3-15
- INT 3-5, 3-11, 3-15

## Integer

- constant 1-14
- division 1-20
- expressions 2-24, 3-1
- results 3-1
- space requirements 1-17
- variable 1-16
- Intrinsic function 1-24, 3-1
- IOCTL 2-51, 3-16
- IOCTL\$ 2-51, 3-16
- I/O functions B-2
- I/O unit, communication B-1

## K

### key(s)

- ALT 1-32, 1-33, 2-79
- BACKSPACE 1-2, 1-32, 1-36
- CAPS LOCK 1-32, 2-79
- combinations 1-34
- CTRL 1-32, 1-33, 1-35, 2-79
- CTRL-ALT-DEL 1-34, 2-80
- CTRL-BREAK 1-4, 1-34, 1-36,  
2-80
- CTRL-END 1-9, 1-36
- CTRL-HOME 1-35
- CTRL-NUM LCK 1-7, 1-34
- CTRL-Z 1-5, 2-90
- DEL 1-36
- DOWN ARROW 1-35
- END 1-35, 1-36
- ESC 1-36
- function 1-31, 1-33, 2-53, 2-54
- HOME 1-35
- INS 1-36
- NUM LCK 1-7, 1-34, 2-79
- Numeric keypad 1-31, 1-34
- Program editor 1-35
- PRTSC 1-32, 2-79
- RETURN 1-9, 1-31
- Scan codes J-1
- SHIFT 1-31, 1-32, 1-33, 2-79
  - Left 2-79
  - Right 2-79

- Symbols 1-32
- TAB 1-36, 2-90
- UP ARROW 1-35
- KEY 2-52, 2-75, 2-78
- LIST 2-52
- [n] OFF 2-54
- [n] ON 2-54
- [n] STOP 2-54
- number 2-52
- OFF 2-52
- ON 2-52
- KEY [n] OFF 2-54
- KEY [n] ON 2-54
- KEY [n] STOP 2-54
- Key number 2-78
- Keyboard 1-31, 2-146
- Keying error 1-6
- Keypad, numeric 1-34, J-2
- KILL 2-56, 2-73, A-2
- KYBD, see keyboard

## L

- LCOPY 2-57
- LEFT\$ 3-17, 3-26
- LEN 3-15, 3-17
- Length 2-5
- LET 2-37, 2-58, 3-10
- LINE 2-27, 2-42, 2-59, 2-92
- Line format 1-11
- LINE INPUT 2-61
- LINE INPUT# 2-41, 2-61, 2-62, 2-87, 2-89
- Line number 1-6, 1-11, 2-2, 2-7, 2-30, 2-44, 2-75, 2-77, 3-10
- LIST 1-6, 1-7, 1-8, 1-9, 2-30, 2-63, 2-128, 2-65, 2-146
- Literals, string 3-15
- LLIST 2-65, 2-146
- LOAD 1-7, 2-66, 2-128, A-1
- LOC 3-18
- LOCATE 1-28, 2-67, 3-6
- LOF 3-19
- LOG 1-2, 3-20
- Logical
  - line 1-36
  - operations 1-11
  - operators 1-21
- Loop 2-143, 2-144, 3-31, 3-33
  - infinite 2-143
- LPOS 3-20, 3-26
- LPRINT 2-68, 3-34
- LPRINT USING 2-68
- LSET 2-37, 2-69, 3-22

## M

- Machine infinity 3-11
- MAT functions E-2
- Mathematical functions G-1
- Maximum block size 1-3
- Medium resolution 1-29, 1-30, 2-145
- Memory allocation C-1
- Memory location 1-3
- MERGE 2-7, 2-70, 2-128, A-2
- Messages, error F-1
- MID\$ 2-71, 3-17, 3-21, 3-26
- MKDIR 2-72
- MKD\$ 3-7, 3-22
- MKI\$ 3-7, 3-22
- MKS\$ 3-7, 3-22
- MOD, see Modulo arithmetic
- Mode,
  - batch 1-2
  - command 1-10
  - direct 1-10
  - graphics 1-28, 1-29, 2-145
  - indirect 1-10
  - program 1-10
  - screen 2-129
- Modes of operation 1-5, 1-10
- Modulo arithmetic 1-20
- Monochrome display 1-28
- MS-DOS 1-1, 2-126, 2-131
  - directory paths 1-27
  - file system 1-26
- Multiple
  - assignments E-2
  - statements E-2
- Music 2-100

## N

- Name 1-27, 2-21
- NAME 2-73
- ...AS A-2
- Nested loops 2-39
- Nesting IF statements 2-46
- NEW 1-6, 2-12, 2-74, 2-141
- Numeric
  - characters 1-11
  - constant 1-14
  - constant type conversion 1-17
  - expressions 3-1
  - fields 2-109
  - keypad 1-9, 1-34
  - values 3-7, 3-22
  - variable names 1-16
- Numerical value 3-2

## O

### Octal

constant 1-14

value 3-22

OCT\$ 3-22

Ok prompt 1-5, 1-10

Offset 2-4

### ON

COM 2-75, 2-83, 2-87

ERROR GOTO 2-76

...GOSUB 2-77, 2-85

...GOTO 2-77

KEY 2-55, 2-78

PEN 2-81, 2-83, 2-99

PLAY 2-82, 2-83

STRIG 2-83

TIMER 2-83, 2-85

OPEN 2-12, 2-37, 2-73, 2-86,

2-90, 2-146, 2-150, A-7

COM 2-87, 2-88, 2-90

### Operations,

arithmetic 1-18, 1-19

logical 1-21

relational 1-18

string 1-24

### Operators,

arithmetic 1-18, 1-19

functional 1-18, 1-24

integer division 1-20

logical 1-18, 1-21, 1-22, 1-23

modulo arithmetic 1-20

relational 1-18, 1-21

OPTION BASE 2-7, 2-26, 2-91

OR 2-115

OUT 2-92, 3-13

Output 2-86, 3-18, 3-19

Overflow 1-20

error 1-18

## P

PAINT 2-93

PALETTE 1-29, 2-96

PALETTE USING 2-97

Parentheses 1-19

Pathname 2-9, 2-72

Paths, directory 1-25, 1-27

PEEK 2-23, 2-103, 3-23

PEN 2-75, 2-78, 2-81, 2-98

Period (.) 1-11, 2-30, 2-101

Pixel 1-28

PLAY 2-100, 3-24, 3-41

PMAP 3-24

POINT 3-25

function 3-25

POKE 2-23, 2-103, 3-23

Port number 2-75

POS 3-6, 3-26

(0) 1-28

PRESET 2-112, 2-115

PRINT 2-68, 2-104, 2-149, 3-34

CHR\$ 2-3

positions 2-68, 2-104

USING 2-106

PRINT# 2-87, 2-110, 2-111,

2-113, 2-150

USING 2-87, 2-110, 2-111,

2-113

### Program

corrections 1-7

editor keys 1-34, 1-35

lines 1-11, 2-121

mode 1-5, 1-6, 1-10

writing, a 1-5

Programming 1-1

### Prompt

Ok 1-5, 1-10

Prompt string 2-48

PRTSC key 1-32

PSET 2-27, 2-112, 2-115

Punctuation Preface-3

PUT 2-37, 2-87, 2-90, B-1

(COM) 2-114

(files) 2-113

(graphics) 2-115

## R

Radius 2-10

### Random

disk files 3-18

File(s) A-7, A-8

file buffer 2-37, 2-41, A-7

RANDOMIZE 2-117

READ 2-118, 2-20

Reference 2-1

Register number 2-96

Relational operators 1-21

Relative form 1-30

REM 2-120

RENUM 2-8, 2-121, 2-122, 3-10

Reserved words 1-12, 1-13

delimiters 1-12

RESET 2-12, 2-122

button 2-143

### Resolution,

high 1-29, 1-30, 2-145

medium 1-29, 1-30, 2-145

RESTORE 2-20, 2-118, 2-123

Results, single precision 3-1

RESUME 2-124

RETURN 2-2, 2-75, 2-125

RIGHT\$ 3-17, 3-26

RMDIR 2-126

RND 2-117, 3-27  
 Root node 1-27  
 RSET 2-12, 2-37, 2-69, 3-22  
 RS-232C 1-2, 1-4, 2-88, 2-90,  
     B-1  
 RUN 1-6, 2-12, 2-117, 2-127,  
     2-148, 3-27, A-1  
  
**S**  
  
 SAVE 1-6, 2-70, 2-128, A-1  
 Scan codes, key J-1  
 Screen  
     editing keys 1-9, 1-10  
     editor 1-9  
     using 1-27  
 SCREEN 1-27, 2-13, 2-59, 2-60,  
     2-98, 2-129, 2-145, 2-146, 3-6,  
     3-27  
 Secondary command processor 2-131  
 Sequential files 3-18, A-2  
 SGN 3-11, 3-28  
 SHELL 2-33, 2-131  
 SHIFT keys 1-31, 1-32, 1-33 2-79  
     Left 2-79  
     Right 2-79  
 Simple variables 3-39  
 SIN 1-2, 3-29  
 Single precision  
     constants 1-15  
     conversion 1-17  
     results 3-1  
     space requirements 1-17  
     variable 1-16  
 SOUND 2-134  
 SPACE\$ 3-29, 3-30  
 SPC 3-29, 3-30  
 Space requirements 1-17  
 Special character(s) 1-12, 3-4  
 SQR 1-2, 3-30  
 Square brackets Preface-3  
 Standard Input 1-2, 1-4  
 Standard Output 1-2, 1-4  
 Start options 1-2  
 Statements,  
     list of I-1, I-3  
     multiple E-2  
 Stdaux 1-3  
 Stderr 1-3  
 Stdin 1-2, 1-3  
 Stdout 1-2, 1-3  
 Stdprn 1-3  
 STICK 3-31  
 STOP 2-19, 2-31, 2-44, 2-135  
 STRIG 2-136, 3-33  
 STRIG(n) 2-83, 2-84, 2-136, 3-33  
     ON/OFF/STOP 2-83, 2-84, 2-136,  
     3-33

**String**  
     constant(s) 1-14, C-4, C-6  
     dimension E-1  
     expressions 2-32, 3-1  
     fields 2-106  
     literals 3-15  
     null 3-8  
     operations 1-24  
     space 1-17, 2-37, A-7  
     values 3-7, 3-22  
     variable 2-37, 3-15, A-7  
     variable names 1-16  
 STRING\$ 3-34, 3-41  
 STR\$ 3-34, 3-38  
 SWAP 2-137  
 Symbols 1-32  
**Syntax**  
     definition 2-1  
     error 2-76  
     list I-1  
     notation Preface-3  
 SYSTEM 1-1, 1-2, 2-12, 2-138

## T

Tab key 1-36  
 TAB 3-34  
 TAN 1-2, 3-35  
 Text 1-27  
     mode 1-29  
 THEN 2-46  
 TIMER 2-117, 2-140, 3-37  
     ON/OFF/STOP 2-117, 2-140, 3-37  
 TIMES\$ 2-139, 3-36  
 Transcendental function 3-1  
 TROFF 2-141  
 TRON 2-141  
 Type conversion 1-17  
 Type declaration characters 1-16

## U

UPARROW key 1-35  
 Uppercase 1-32  
 User-defined functions 1-16,  
     1-24  
 USR 2-23, 3-37, C-5

## V

VAL 3-32, 3-38

### Value(s)

absolute 3-2  
attribute 3-25  
floating point 3-1  
hex 3-12  
numeric 3-7  
octal 3-22  
string 3-7, 3-22

### Variable(s),

array 1-16, 1-17  
double precision 1-16, 2-137  
integer 1-16, 2-137  
list of I-1, I-9  
simple 3-39  
single precision 1-16, 2-137  
string 2-48, 3-15, A-7

### Variable names

numeric 1-15, 1-16  
string 1-15, 1-16  
VARPTR 2-23, 3-39  
VARPTR\$ 3-41  
VIEW 2-142, 2-147  
Vpage 2-129

## W

WAIT 2-143  
WHILE 2-125  
WHILE...WEND 2-144  
Width, infinite 2-146  
WIDTH 1-28, 2-145, 3-27  
WINDOW 2-147  
Word 1-35  
World coordinates 2-147  
WRITE 2-149  
WRITE# 2-111, 2-113, 2-150

## X

Xoffset 1-30  
XOR 2-115, 2-116

## Y

Yoffset 1-30

[illegible]



[illegible]

[illegible]

[illegible]



